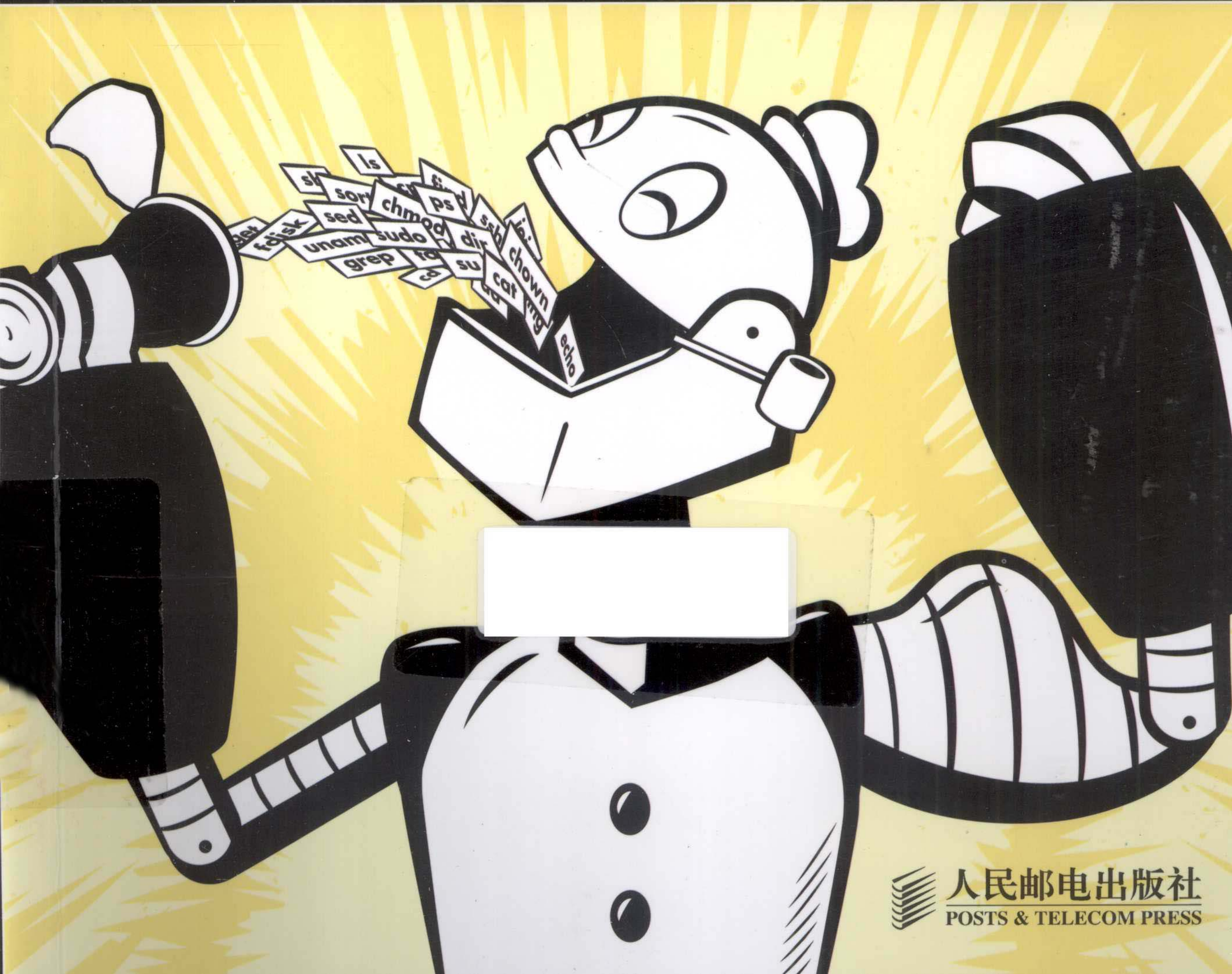
no starch press

Linux命令行大全

THE LINUX COMMAND LINE

[美] William E. Shotts, Jr. 著　　郭光伟　郝记生 译



人民邮电出版社
POSTS & TELECOM PRESS

Linux命令行大全

THE LINUX COMMAND LINE

[美] William E. Shotts, Jr. 著　　郭光伟　郝记生 译

人 民 邮 电 出 版 社
北 京

图书在版编目（C I P）数据

Linux命令行大全 / （美）绍茨（Shotts,W.E.）著；郭光伟，郝记生译. -- 北京：人民邮电出版社，2013.3
ISBN 978-7-115-30745-3

Ⅰ. ①L… Ⅱ. ①绍… ②郭… ③郝… Ⅲ. ①Linux操作系统—程序设计 Ⅳ. ①TP316.89

中国版本图书馆CIP数据核字(2013)第003381号

版权声明

Copyright © 2012 by William E. Shotts, Jr. Title of English-language original: The Linux Command Line: A Complete Introduction, ISBN 978-1-59327-389-7, published by No Starch Press. Simplified Chinese-language edition copyright © 2012 by Posts and Telecom Press. All rights reserved.

本书中文简体字版由美国 No Starch 出版社授权人民邮电出版社出版。未经出版者书面许可，对本书任何部分不得以任何方式复制或抄袭。

版权所有，侵权必究。

Linux 命令行大全

◆ 著　[美] William E.Shotts,Jr
译　郭光伟　郝记生
责任编辑　傅道坤

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街 14 号
邮编　100061　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京昌平百善印刷厂印刷

◆ 开本：800×1000　1/16
印张：28.25
字数：677 千字　2013 年 3 月第 1 版
印数：1 – 3 000 册　2013 年 3 月北京第 1 次印刷

著作权合同登记号　图字：01-2012-2972 号

ISBN 978-7-115-30745-3

定价：69.00 元

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154

内 容 提 要

本书主要介绍 Linux 命令行的使用，循序渐进，深入浅出，引导读者全面掌握命令行的使用方法。

本书分为四部分。第一部分开始了对命令行基本语言的学习之旅，包括命令结构、文件系统的导引、命令行的编辑以及关于命令的帮助系统和使用手册。第二部分主要讲述配置文件的编辑，用于计算机操作的命令行控制。第三部分讲述了从命令行开始执行的常规任务。类 UNIX 操作系统，比如 Linux，包含了很多“经典的”命令行程序，这些程序可以高效地对数据进行操作。第四部分介绍了 shell 编程，这是一个公认的初级技术，并且容易学习，它可以使很多常见的系统任务自动运行。通过学习 shell 编程，读者也可以熟悉其他编程语言的使用。

本书适合从其他平台过渡到 Linux 的新用户和初级 Linux 服务器管理员阅读。没有任何 Linux 基础和 Linux 编程经验的读者，也可以通过本书掌握 Linux 命令行的使用方法。

致　　谢

我要感谢以下朋友帮助促成此书。

首先要感谢的是启发我的人：Wiley 出版社的组稿编辑 Jenney Watson，是她最初建议我写一本与 shell 脚本相关的书。尽管 Wiley 出版社最终没有接受我的书稿，但它成为本书的基础。John C. Dvorak，一位著名的专栏作家和权威人士，给了我重要的建议。在他的视频播客“Cranky Geeks”的一个片段中，他描述了这样一个写作过程：“哇喔！每天写 200 字，一年之后，你将拥有一部小说”。这个建议促使我每天写一页，直到完成本书。Dmitri Popov 在 *Free Software Magazine* 写了一篇名为 *Creating a book template with Writer* 的文章，它启发我使用 OpenOffice.org Writer 来排版文字，事实证明，结果还不错。

其次感谢那些帮助我制作本书最初的、可自由分发的电子版本（可在 LinuxCommand. org 找到）的志愿者：Mark Polesky 非常出色地审阅和检验了本书的文字；Jesse Becker、Tomasz Chrzczonowicz、Michael Levin 和 Spence Miner 也审阅和复审了部分文字；Karen M. Shotts 花了很多时间编辑我的书稿。

还要感谢在 No Starch 出版社长时间辛苦工作，保证本书出版、正常发售的朋友，他们是产品经理 Serena Yang、编辑 Keith Fancher 和 No Starch 出版社的其他员工。

最后要感谢 LinuxCommand.org 的读者，他们给我发了很多有价值的邮件。是他们的鼓舞让我感觉到，我真的是在做一件非比寻常的事情。

前　言

我想给大家讲一个故事。故事内容不是 Linus Torvalds 在 1991 年怎样编写了 Linux 内核的第一个版本，因为这些内容你可以在很多 Linux 图书中找到。我也不想告诉你，更早之前，Richard Stallman 是如何开始 GNU 项目，设计了一个免费的类 UNIX 操作系统。那也是一个很有意义的故事，但大多数 Linux 图书也讲到了它。我想给大家讲一个如何才能夺回计算机控制权的故事。

在 20 世纪 70 年代后期，我刚开始和计算机打交道时，正在进行着一场革命，那时的我还是一名大学生。微处理器的发明使得你我这样的普通人真正拥有一台计算机成为可能。今天，人们难以想象，只有大公司和强大的政府机构才能够使用计算机的世界，是怎样的一个世界。让我说，你其实想象不出多少来。

如今，世界已经截然不同。计算机遍布各个领域，从小手表到大型数据中心，以及介于它们之间的每一样东西。除了随处可见的计算机之外，我们还有一个无处不在的连接所有计算机的网络。这开创了一个奇妙的个人授权和创作自由的新时代。但是在过去的二三十年里，一些事情在悄然发生。一个大公司不断地把它的控制权强加到世界绝大多数的计算机上，并且决定你对计算机的操作权力。幸运的是，世界各地的人们正在努力进行抗争。他们通过自己编写软件来争夺自己计算机的控制权。他们创造了 Linux！

很多人提到 Linux 的时候都会讲到“自由”，但是并不是所有人都明白这种自由到底意味着什么。自由就是能够决定计算机可以做什么，而获得这种自由的唯一方法就是知道你的计算机正在做什么；自由就是计算机没有秘密可言，

只要你仔细地寻找，就能了解其全部内容。

为什么使用命令行

读者之前应该注意到，电影中的“超级黑客”，就是那些能够在 30 秒内入侵到超级安全的军方计算机里的家伙，都是坐在计算机旁，从来不碰鼠标的。这是因为电影制片人意识到，我们人类从本能上会明白，能够让计算机执行任何任务的唯一途径，是通过键盘输入命令来实现的。

现在，大多数计算机用户只熟悉图形用户界面（GUI），并且产品供应商和专家还在不停地灌输一种思想，那就是命令行界面（CLI）是一种很糟糕的东西，而且已经过时。这是很不幸的，因为一个好的命令行界面是一种很神奇的人机交互方式，就和我们采用书信进行交流一样。据说“图形用户界面能让简单的任务更简单，而命令行界面能够处理复杂的任务”，这句话在今天看来仍然是正确的。

由于 Linux 系统参照了 UNIX 系列操作系统，它分享了 UNIX 系统丰富的命令行工具。UNIX 系统在 20 世纪 80 年代早期就占据了主流地位（尽管它只是在 20 世纪 70 年代才开发出来），结果，在普遍采用图形用户界面之前，开发了一种广泛使用的命令行界面。事实上，Linux 开发者优先使用命令行界面（而不是其他系统，比如 Windows NT）的一个原因就是因为其强大的命令行界面，使“完成复杂的任务成为可能”。

本书内容

这是一本全面讲述如何使用 Linux 命令行的图书。与那些仅涉及一个程序（比如 shell 程序、bash）的图书不同，本书从更广泛的意义上向读者传授如何使用命令行，它是如何工作的，它有哪些功能，以及使用它的最佳方式是什么。

这不是一本关于 Linux 系统管理方面的图书。任何一个关于命令行的重要讨论，都一定会涉及系统管理方面的内容，但是本书只涉及很少的管理方面的

问题。本书为读者准备了其他的学习内容，帮助你为使用命令行打下坚实的基础，这可是完成一个系统管理任务所必需的至关重要的工具。

本书以 Linux 为中心。其他许多图书为了扩大读者群体以及自身的影响力，会在书中包含其他平台，比如通用的 UNIX 和 Mac OS X 系统。而且为了达到这个目的，它们只能“淡化”书的内容，只讲解一些通用的主题。而本书只包括当前的 Linux 发行版本。尽管本书 95%的内容对其他类 UNIX 系统用户也有帮助，但是本书主要还是面向现代的 Linux 命令行用户。

本书读者对象

本书适合从其他平台转到 Linux 的新用户阅读。这些新用户很可能原来是 Microsoft Windows 版本的超级用户；也可能是老板已经要求负责管理一个 Linux 服务器的管理员；还有可能是厌烦了桌面系统的安全问题，想要体验一下 Linux 系统的用户。没关系，不管你属于哪类用户，都欢迎阅读本书。

不过一般来说，Linux 的启蒙学习不存在任何捷径。命令行的学习具有挑战性而且颇费精力，这倒不是因为它太难，而是它涵盖的内容太多。一般的 Linux 系统有上千个程序可以通过命令行使用，这点毫不夸张。你需要提醒自己的是，命令行可不是随便就能学会的。

另一方面，学习 Linux 命令行也非常值得。如果你认为自己已经是一名“超级用户”了，那么请注意，你可能不知道什么才是真正的“超级用户”。不同于许多其他的计算机技术，命令行的知识是经久不衰的。今天学会的技能，在 10 年后仍然有用。换言之，命令行是能够历经时间考验的。

如果读者没有编程经验，也不用担心，你仍然可以从本书开始学习。

内容安排

本书精心编排，内容有序，就像有一位老师坐在你身旁，耐心指导你。许多作者都采用系统化的方法来讲解本书中的内容。对作者来讲，这很合理，但

是对初学者来讲，则可能摸不着头脑。

本书的另一个目的是使读者熟悉 UNIX 的思考方式，它与 Windows 的思考方式大不相同。在学习的过程中，本书还将帮助读者理解命令行的工作原理和方式。Linux 不仅仅是一个软件，它还是庞大的 UNIX 文化中的一小部分，有着自己的语言和历史。同时，我也许会说一些过激的言语。

本书分为四部分，每一部分都讲解了不同方面的命令行知识。

- **第一部分：学习 shell**。开始对命令行基本语言的学习之旅，包括命令结构、浏览文件系统、编辑命令行，以及查找命令帮助文档。
- **第二部分：配置和环境**。这部分主要讲述如何编写配置文件，通过配置文件，用命令行的方式控制计算机操作。
- **第三部分：常见任务和主要工具**。这部分讲述了许多通过命令行来执行的常规任务。类 UNIX 操作系统，比如 Linux，包含了很多“经典的”命令行程序，这些程序可以对数据进行强大的操作。
- **第四部分：编写 shell 脚本**。这部分介绍了 shell 编程，这是一个公认的基本技能，它很容易学习，而且它可以自动执行很多常见的计算任务。通过学习 shell 编程，你会逐渐熟悉一些关于编程语言方面的概念，这些概念也适用于其他编程语言。

如何阅读

建议读者从头到尾地阅读本书。本书并不是一本技术参考手册，实际上它更像一本故事书：有开头，有过程，有结尾。

预备知识

为了使用本书，你需要安装 Linux 操作系统。你可以通过两种方式来完成安装。

- 在计算机（不需要是最新的）上安装 Linux 系统。选择哪个 Linux 发行版本没有关系，虽然现在大多数人在开始的时候会选择 Ubuntu、Fedora，或者是 OpenSUSE。如果你拿不准，那就先试试 Ubuntu。由于主机硬件配置不同，安装 Linux 时，你可能不费吹灰之力就安装上了，也可能费了九牛二虎之力还安装不上。本书建议使用一台几年前的台式计算机，并且有至少 256MB 的内存和 6GB 的空闲磁盘空间。尽可能避免使用笔记本计算机和无线网络，在 Linux 环境下，它们经常不能工作。

- 使用 live CD。许多 Linux 发行版本都自带一个比较酷的功能，你可以直接从 CD-ROM 中运行 Linux，而不必安装 Linux。只需进入 BIOS 设置界面，设置计算机从 CD-ROM 启动，插入一个 live CD，然后重新启动。在进行安装之前，使用 live CD 可以检测计算机对 Linux 的兼容性。但是缺点是会比通过硬盘安装 Linux 要慢好多。Ubuntu 和 Fedora 都有 live CD 版本。

注意 无论采用何种方式来安装 Linux，都将需要有临时的超级用户（也就是管理员）特权来执行本书中所讲的内容。

在安装好 Linux 系统之后，就一边开始阅读本书，一边练习吧。本书大部分内容都可以自己动手练习，所以坐下来，开始敲入命令并体验吧。

本书为什么不称为“GNU/LINUX”

在某些领域，称 Linux 操作系统为“GNU/LINUX 操作系统”在政治立场上是正确的。没有一个完全正确的方式能命名它，因为它是由许多分布在世界各地的贡献者们合作开发而成的。从技术层面来讲，Linux 只是操作系统的内核名字，没有别的含义。当然，内核非常重要，因为有了它，操作系统才能运行起来，但是它不能构成一个完备的操作系统。

Richard Stallman 是一位天才的哲学家、自由软件运动的创始人、自由基金会创办者，他创建了自由软件项目，编写了 GNU C 语言编译器（GCC）的第一个版本，并且创建了 GNU 公用许可证（GPL）等。他坚持将 Linux 称为“GNU/LINUX”，为的是准确地反映 GNU 项目对 Linux 操作系统所做出的贡献。尽管 GNU 项目先于 Linux 内核出现，并且这个项目所做出的贡献得到了极高的赞誉，但是将 GNU 加入 Linux 名字里面则对其他每一个为 Linux

的发展做出巨大贡献的人们来说，就不公平了。除此之外，由于计算机在运行的时候首先启动内核，其他的所有软件都在其之上运行，所以本书认为“Linux/GNU”这个名字在技术上来讲更精准一些。

在目前流行的用法中，“Linux”指的是内核以及在一个典型的 Linux 发行版本中所包含的所有免费和开源软件，也就是说，整个 Linux 体系并非仅有 GNU 项目软件。在操作系统业界，好像更钟情于单个词的名字，比如 DOS、Windows、Solaris、Iris 和 AIX。所以本书选择使用流行的命名规则。然而，如果读者倾向于使用“GNU/LINUX”，那么在阅读本书时，可以搜索并替换“Linux”，我不会介意的。

目　　录

第一部分　学习 shell

第 1 章　shell 是什么……3
　1.1　终端仿真器……3
　1.2　第一次键盘输入……4
　　1.2.1　命令历史记录……4
　　1.2.2　光标移动……4
　1.3　几个简单的命令……5
　1.4　结束终端会话……6
第 2 章　导航……7
　2.1　理解文件系统树……7
　2.2　当前工作目录……8
　2.3　列出目录内容……9
　2.4　更改当前工作目录……9
　　2.4.1　绝对路径名……9
　　2.4.2　相对路径名……9
　　2.4.3　一些有用的快捷方式……10
第 3 章　Linux 系统……13
　3.1　ls 命令的乐趣……13

3.1.1 选项和参数 …… 14
3.1.2 进一步了解长列表格式 …… 15
3.2 使用 file 命令确定文件类型 …… 16
3.3 使用 less 命令查看文件内容 …… 16
3.4 快速浏览 …… 18
3.5 符号链接 …… 20
第 4 章 操作文件与目录 …… 23
4.1 通配符 …… 24
4.2 mkdir——创建目录 …… 26
4.3 cp——复制文件和目录 …… 26
4.4 mv——移除和重命名文件 …… 27
4.5 rm——删除文件和目录 …… 28
4.6 ln——创建链接 …… 29
4.6.1 硬链接 …… 29
4.6.2 符号链接 …… 30
4.7 实战演练 …… 30
4.7.1 创建目录 …… 30
4.7.2 复制文件 …… 31
4.7.3 移动和重命名文件 …… 31
4.7.4 创建硬链接 …… 32
4.7.5 创建符号链接 …… 33
4.7.6 移除文件和目录 …… 34
4.8 本章结尾语 …… 35
第 5 章 命令的使用 …… 37
5.1 究竟什么是命令 …… 38
5.2 识别命令 …… 38
5.2.1 type——显示命令的类型 …… 38
5.2.2 which——显示可执行程序的位置 …… 39
5.3 获得命令文档 …… 39
5.3.1 help——获得 shell 内置命令的帮助文档 …… 39
5.3.2 help——显示命令的使用信息 …… 40
5.3.3 man——显示程序的手册页 …… 40
5.3.4 apropos——显示合适的命令 …… 41

5.3.5 whatis——显示命令的简要描述 …… 42
5.3.6 info——显示程序的 info 条目 …… 42
5.3.7 README 和其他程序文档文件 …… 43
5.4 使用别名创建自己的命令 …… 43
5.5 温故以求新 …… 45
第 6 章 重定向 …… 47
6.1 标准输入、标准输出和标准错误 …… 48
6.1.1 标准输出重定向 …… 48
6.1.2 标准错误重定向 …… 50
6.1.3 将标准输出和标准错误重定向到同一个文件 …… 50
6.1.4 处理不想要的输出 …… 51
6.1.5 标准输入重定向 …… 51
6.2 管道 …… 53
6.2.1 过滤器 …… 53
6.2.2 uniq——报告或忽略文件中重复的行 …… 54
6.2.3 wc——打印行数、字数和字节数 …… 54
6.2.4 grep——打印匹配行 …… 54
6.2.5 head/tail——输出文件的开头部分/结尾部分 …… 55
6.2.6 tee——从 stdin 读取数据，并同时输出到 stdout 和文件 …… 56
6.3 本章结尾语 …… 57
第 7 章 透过 shell 看世界 …… 59
7.1 扩展 …… 59
7.1.1 路径名扩展 …… 60
7.1.2 波浪线扩展 …… 61
7.1.3 算术扩展 …… 61
7.1.4 花括号扩展 …… 62
7.1.5 参数扩展 …… 63
7.1.6 命令替换 …… 64
7.2 引用 …… 65
7.2.1 双引号 …… 65
7.2.2 单引号 …… 67
7.2.3 转义字符 …… 67
7.3 本章结尾语 …… 68

第 8 章 高级键盘技巧……69
 8.1 编辑命令行……69
  8.1.1 光标移动……70
  8.1.2 修改文本……70
  8.1.3 剪切和粘贴（Killing and Yanking）文本……71
 8.2 自动补齐功能……71
 8.3 使用历史命令……73
  8.3.1 搜索历史命令……73
  8.3.2 历史记录扩展……75
 8.4 本章结尾语……76
第 9 章 权限……77
 9.1 所有者、组成员和其他所有用户……78
 9.2 读取、写入和执行……79
  9.2.1 chmod——更改文件模式……81
  9.2.2 采用 GUI 设置文件模式……84
  9.2.3 umask——设置默认权限……85
 9.3 更改身份……87
  9.3.1 su——以其他用户和组 ID 的身份来运行 shell……88
  9.3.2 sudo——以另一个用户的身份执行命令……89
  9.3.3 chown——更改文件所有者和所属群组……90
  9.3.4 chgrp——更改文件所属群组……91
 9.4 权限的使用……91
 9.5 更改用户密码……93
第 10 章 进程……95
 10.1 进程如何工作……96
  10.1.1 使用 ps 命令查看进程信息……96
  10.1.2 使用 top 命令动态查看进程信息……98
 10.2 控制进程……100
  10.2.1 中断进程……100
  10.2.2 使进程在后台运行……101
  10.2.3 使进程回到前台运行……101
  10.2.4 停止（暂停）进程……102
 10.3 信号……102

10.3.1 使用 kill 命令发送信号到进程 ········ 103
10.3.2 使用 killall 命令发送信号给多个进程 ········ 105
10.4 更多与进程相关的命令 ········ 105

第二部分　配置与环境

第 11 章　环境 ········ 109
11.1 环境中存储的是什么 ········ 109
11.1.1 检查环境 ········ 110
11.1.2 一些有趣的变量 ········ 111
11.2 环境是如何建立的 ········ 112
11.2.1 login 和 non-login shell ········ 112
11.2.2 启动文件中有什么 ········ 113
11.3 修改环境 ········ 114
11.3.1 用户应当修改哪些文件 ········ 114
11.3.2 文本编辑器 ········ 115
11.3.3 使用文本编辑器 ········ 115
11.3.4 激活我们的修改 ········ 117
11.4 本章结尾语 ········ 118
第 12 章　VI 简介 ········ 119
12.1 为什么要学习 vi ········ 119
12.2 VI 背景 ········ 120
12.3 启动和退出 vi ········ 120
12.4 编辑模式 ········ 121
12.4.1 进入插入模式 ········ 122
12.4.2 保存工作 ········ 122
12.5 移动光标 ········ 123
12.6 基本编辑 ········ 124
12.6.1 添加文本 ········ 124
12.6.2 插入一行 ········ 125
12.6.3 删除文本 ········ 126
12.6.4 剪切、复制和粘贴文本 ········ 127
12.6.5 合并行 ········ 128
12.7 查找和替换 ········ 128

12.7.1 行内搜索……128
12.7.2 搜索整个文件……129
12.7.3 全局搜索和替换……129
12.8 编辑多个文件……130
12.8.1 切换文件……131
12.8.2 载入更多的文件……132
12.8.3 文件之间的内容复制……132
12.8.4 插入整个文件……133
12.9 保存工作……134
第 13 章 定制提示符……135
13.1 提示符的分解……135
13.2 尝试设计提示符……137
13.3 添加颜色……138
13.4 移动光标……140
13.5 保存提示符……141
13.6 本章结尾语……141

第三部分　常见任务和主要工具

第 14 章 软件包管理……145
14.1 软件包系统……146
14.2 软件包系统工作方式……146
14.2.1 软件包文件……146
14.2.2 库……147
14.2.3 依赖关系……147
14.2.4 高级和低级软件包工具……147
14.3 常见软件包管理任务……148
14.3.1 在库里面查找软件包……148
14.3.2 安装库中的软件包……148
14.3.3 安装软件包文件中的软件包……149
14.3.4 删除软件包……149
14.3.5 更新库中的软件包……150
14.3.6 更新软件包文件中的软件包……150
14.3.7 列出已安装的软件包列表……150

14.3.8 判断软件包是否安装 151
14.3.9 显示已安装软件包的相关信息 151
14.3.10 查看某具体文件由哪个软件包安装得到 151
14.4 本章结尾语 152
第 15 章 存储介质 155
15.1 挂载、卸载存储设备 156
15.1.1 查看已挂载的文件系统列表 157
15.1.2 确定设备名称 160
15.2 创建新的文件系统 162
15.2.1 用 fdisk 命令进行磁盘分区 162
15.2.2 用 mkfs 命令创建新的文件系统 164
15.3 测试、修复文件系统 165
15.4 格式化软盘 166
15.5 直接从/向设备转移数据 166
15.6 创建 CD-ROM 映像 167
15.6.1 创建一个 CD-ROM 文件映像副本 167
15.6.2 从文件集合中创建映像文件 168
15.7 向 CD-ROM 写入映像文件 168
15.7.1 直接挂载 ISO 映像文件 168
15.7.2 擦除可读写 CD-ROM 169
15.7.3 写入映像文件 169
15.8 附加认证 169
第 16 章 网络 171
16.1 检查、监测网络 172
16.1.1 ping——向网络主机发送特殊数据包 172
16.1.2 traceroute——跟踪网络数据包的传输路径 173
16.1.3 netstat——检查网络设置及相关统计数据 174
16.2 通过网络传输文件 175
16.2.1 ftp——采用 FTP（文件传输协议）传输文件 175
16.2.2 lftp——更好的 ftp（文件传输协议） 177
16.2.3 wget——非交互式网络下载工具 177
16.3 与远程主机的安全通信 178
16.3.1 ssh——安全登录远程计算机 178

16.3.2 scp 和 sftp——安全传输文件 …… 181
第 17 章 文件搜索 …… 183
17.1 locate——较简单的方式查找文件 …… 184
17.2 find——较复杂的方式查找文件 …… 185
17.2.1 test 选项 …… 186
17.2.2 action 选项 …… 190
17.2.3 返回到 playground 文件夹 …… 194
17.2.4 option 选项 …… 196
第 18 章 归档和备份 …… 197
18.1 文件压缩 …… 198
18.1.1 gzip——文件压缩与解压缩 …… 198
18.1.2 bzip2——牺牲速度以换取高质量的数据压缩 …… 200
18.2 文件归档 …… 201
18.2.1 tar——磁带归档工具 …… 201
18.2.2 zip——打包压缩文件 …… 205
18.3 同步文件和目录 …… 207
18.3.1 rsync——远程文件、目录的同步 …… 207
18.3.2 在网络上使用 rsync 命令 …… 209
第 19 章 正则表达式 …… 211
19.1 什么是正则表达式 …… 211
19.2 grep——文本搜索 …… 212
19.3 元字符和文字 …… 213
19.4 任意字符 …… 214
19.5 锚 …… 214
19.6 中括号表达式和字符类 …… 215
19.6.1 否定 …… 216
19.6.2 传统字符范围 …… 216
19.6.3 POSIX 字符类 …… 217
19.7 POSIX 基本正则表达式和扩展正则表达式的比较 …… 220
19.8 或选项 …… 221
19.9 限定符 …… 222
19.9.1 ? ——匹配某元素 0 次或 1 次 …… 222
19.9.2 *——匹配某元素多次或零次 …… 222

19.9.3 +——匹配某元素一次或多次……223
19.9.4 {}——以指定次数匹配某元素……223
19.10 正则表达式的应用……224
19.10.1 用 grep 命令验证号码簿……224
19.10.2 用 find 查找奇怪文件名的文件……225
19.10.3 用 locate 查找文件……226
19.10.4 利用 less 和 vim 命令搜索文本……226
19.11 本章结尾语……227
第 20 章 文本处理……229
20.1 文本应用程序……230
20.1.1 文件……230
20.1.2 网页……230
20.1.3 电子邮件……230
20.1.4 打印机输出……231
20.1.5 程序源代码……231
20.2 温故以求新……231
20.2.1 cat——进行文件之间的拼接并且输出到标准输出……231
20.2.2 sort——对文本行进行排序……232
20.2.3 uniq——通知或省略重复的行……238
20.3 切片和切块……239
20.3.1 cut——删除文本行中的部分内容……239
20.3.2 paste——合并文本行……242
20.3.3 join——连接两文件中具有相同字段的行……243
20.4 文本比较……245
20.4.1 comm——逐行比较两个已排序文件……245
20.4.2 diff——逐行比较文件……246
20.4.3 patch——对原文件进行 diff 操作……248
20.5 非交互式文本编辑……249
20.5.1 tr——替换或删除字符……249
20.5.2 sed——用于文本过滤和转换的流编辑器……251
20.5.3 aspell——交互式拼写检查工具……258
20.6 本章结尾语……260
20.7 附加项……261

第 21 章 格式化输出……263
21.1 简单的格式化工具……264
21.1.1 nl——对行进行标号……264
21.1.2 fold—将文本中的行长度设定为指定长度……266
21.1.3 fmt——简单的文本格式化工具……267
21.1.4 pr——格式化打印文本……270
21.1.5 printf——格式化并打印数据……270
21.2 文档格式化系统……273
21.2.1 roff 和 T_EX 家族……274
21.2.2 groff——文档格式化系统……274
21.3 本章结尾语……279
第 22 章 打印……281
22.1 打印操作简史……282
22.1.1 灰暗时期的打印……282
22.1.2 基于字符的打印机……282
22.1.3 图形化打印机……283
22.2 Linux 方式的打印……284
22.3 准备打印文件……284
22.3.1 pr——将文本文件转换为打印文件……285
22.4 向打印机发送打印任务……285
22.4.1 lpr——打印文件（Berkeley 类型）……286
22.4.2 lp——打印文件（System V 类型）……287
22.4.3 另外一个参数选项：a2ps……287
22.5 监测和控制打印任务……290
22.5.1 lpstat——显示打印系统状态……290
22.5.2 lpq——显示打印队列状态……291
22.5.3 lprm 与 cancel——删除打印任务……291
第 23 章 编译程序……293
23.1 什么是编译……294
23.2 是不是所有的程序都需要编译……295
23.3 编译一个 C 程序……295
23.3.1 获取源代码……296
23.3.2 检查源代码树……297

23.3.3 生成程序 …… 298
23.3.4 安装程序 …… 302
23.4 本章结尾语 …… 302

第四部分 编写 shell 脚本

第 24 章 编写第一个 shell 脚本 …… 305
24.1 什么是 shell 脚本 …… 305
24.2 怎样写 shell 脚本 …… 306
24.2.1 脚本文件的格式 …… 306
24.2.2 可执行权限 …… 307
24.2.3 脚本文件的位置 …… 307
24.2.4 脚本的理想位置 …… 308
24.3 更多的格式诀窍 …… 309
24.3.1 长选项名 …… 309
24.3.2 缩进和行连接 …… 309
24.5 本章结尾语 …… 310
第 25 章 启动一个项目 …… 311
25.1 第一阶段：最小的文档 …… 311
25.2 第二阶段：加入一点数据 …… 313
25.3 变量和常量 …… 314
25.3.1 创建变量和常量 …… 314
25.3.2 为变量和常量赋值 …… 316
25.4 here 文档 …… 317
25.5 本章结尾语 …… 319
第 26 章 自顶向下设计 …… 321
26.1 shell 函数 …… 322
26.2 局部变量 …… 325
26.3 保持脚本的运行 …… 326
26.4 本章结尾语 …… 328
第 27 章 流控制：IF 分支语句 …… 329
27.1 使用 if …… 330
27.2 退出状态 …… 330

27.3 使用 test 命令 332
27.3.1 文件表达式 332
27.3.2 字符串表达式 334
27.3.3 整数表达式 335
27.4 更现代的 test 命令版本 336
27.5 (( ))——为整数设计 338
27.6 组合表达式 339
27.7 控制运算符：另一种方式的分支 341
27.8 本章结尾语 342
第 28 章 读取键盘输入 343
28.1 read——从标准输入读取输入值 344
28.1.1 选项 346
28.1.2 使用 IFS 间隔输入字段 347
28.2 验证输入 349
28.3 菜单 350
28.4 本章结尾语 351
28.5 附加项 352
第 29 章 流控制：WHILE 和 UNTIL 循环 353
29.1 循环 353
29.2 while 354
29.3 跳出循环 356
29.4 until 357
29.5 使用循环读取文件 358
29.6 本章结尾语 358
第 30 章 故障诊断 359
30.1 语法错误 359
30.1.1 引号缺失 360
30.1.2 符号缺失冗余 360
30.1.3 非预期的展开 361
30.2 逻辑错误 362
30.2.1 防御编程 363
30.2.2 输入值验证 364
30.3 测试 364

30.3.1 桩 ······ 365
30.3.2 测试用例 ······ 365
30.4 调试 ······ 366
30.4.1 找到问题域 ······ 366
30.4.2 追踪 ······ 366
30.4.3 运行过程中变量的检验 ······ 368
30.5 本章结尾语 ······ 369
第 31 章 流控制：case 分支 ······ 371
31.1 case ······ 371
31.1.1 模式 ······ 373
31.1.2 多个模式的组合 ······ 374
31.2 本章结尾语 ······ 375
第 32 章 位置参数 ······ 377
32.1 访问命令行 ······ 377
32.1.1 确定实参的数目 ······ 378
32.1.2 shift——处理大量的实参 ······ 379
32.1.3 简单的应用程序 ······ 380
32.1.4 在 shell 函数中使用位置参数 ······ 381
32.2 处理多个位置参数 ······ 381
32.3 更完整的应用程序 ······ 383
32.4 本章结尾语 ······ 386
第 33 章 流控制：for 循环 ······ 389
33.1 for：传统 shell 形式 ······ 389
33.2 for：C 语言形式 ······ 392
33.3 本章结尾语 ······ 393
第 34 章 字符串和数字 ······ 395
34.1 参数扩展（Parameter Expansion） ······ 395
34.1.1 基本参数 ······ 396
34.1.2 空变量扩展的管理 ······ 396
34.1.3 返回变量名的扩展 ······ 397
34.1.4 字符串操作 ······ 398
34.2 算术计算和扩展 ······ 400
34.2.1 数字进制 ······ 401

34.2.2 一元运算符 …… 401
34.2.3 简单算术 …… 401
34.2.4 赋值 …… 402
34.2.5 位操作 …… 404
34.2.6 逻辑操作 …… 405
34.3 bc：一种任意精度计算语言 …… 407
34.3.1 bc 的使用 …… 407
34.3.2 脚本例子 …… 408
34.4 本章结尾语 …… 409
34.5 附加项 …… 409
第 35 章 数组 …… 411
35.1 什么是数组 …… 411
35.2 创建一个数组 …… 412
35.3 数组赋值 …… 412
35.4 访问数组元素 …… 413
35.5 数组操作 …… 414
35.5.1 输出数组的所有内容 …… 415
35.5.2 确定数组元素的数目 …… 415
35.5.3 查找数组中使用的下标 …… 416
35.5.4 在数组的结尾增加元素 …… 416
35.5.5 数组排序操作 …… 416
35.5.6 数组的删除 …… 417
35.6 本章结尾语 …… 418
第 36 章 其他命令 …… 419
36.1 组命令和子 shell …… 419
36.1.1 执行重定向 …… 420
36.1.2 进程替换 …… 420
36.2 trap …… 422
36.3 异步执行 …… 425
36.4 命名管道 …… 426
36.4.1 设置命名管道 …… 427
36.4.2 使用命名管道 …… 427
36.5 本章结尾语 …… 428

第一部分

学习 shell

第 1 章 shell 是什么

当谈到命令行时，我们实际上指的是 shell。shell 是一个接收由键盘输入的命令，并将其传递给操作系统来执行的程序。几乎所有的 Linux 发行版都提供 shell 程序，该程序来自于称之为 bash 的 GNU 项目。bash 是 Bourne Again Shell 的首字母缩写，Bourne Again Shell 基于这样一个事实，即 bash 是 sh 的增强版本，而 sh 是最初的 UNIX shell 程序，由 Steve Bourne 编写。

1.1 终端仿真器

当使用图形用户界面时，需要另一种叫做终端仿真器（terminal emulator）的程序与 shell 进行交互。如果我们仔细查看桌面菜单，那么很可能会找到一款终端仿真器。在 KDE 环境下使用的是 konsole，而在 GNOME 环境下使用的是 gnome-terminal，但是在桌面菜单上很可能将它们简单地统称为终端。在 Linux 系统中，还有许多其他的终端仿真器可以使用，但是它们基本上都做同样的事情：让用户访问 shell。因为不同的终端仿真器所具有的功能特性不尽相同，因

此，你可以根据自己的喜好进行选择。

1.2 第一次键盘输入

现在开始吧。启动终端仿真器！运行后的终端仿真器如下所示。

```
[me@linuxbox ~]$
```

这称为 shell 提示符，只要 shell 准备接受外部输入，它就会出现。在不同的发行版中，提示符的外观可能会有所差异，但是，它通常包括 username@machinename，其后是当前工作目录（长度更长一些）和一个$符号。

如果 shell 提示符的最后一个字符是#，而不是一个$符号，那么终端会话将享有超级用户特权。这就意味着要么我们是以根用户身份登录，要么我们选择的终端仿真器可以提供超级用户（管理）特权。

假定一切工作都很顺利，接下来尝试输入一些内容。在提示符后输入一些乱码，如下所示。

```
[me@linuxbox ~]$ kaekfjaeifj
```

由于这些命令没有任何意义，shell 会让我们重新输入。

```
bash: kaekfjaeifj: command not found
[me@linuxbox ~]$
```

1.2.1 命令历史记录

如果按下向上方向指示键，将会看到先前的命令 kaekfjaeifj 再一次出现在提示符的后面，这称为命令历史记录。在默认情况下，大部分 Linux 发行版本能够存储最近输入的 500 个命令。按下向下方向指示键，则先前的命令消失。

1.2.2 光标移动

再次按下向上方向指示键，重新调用先前的命令，然后分别按下向左和向右方向指示键，看看如何将光标定位到命令行的任意位置。这可以让我们很容易地编辑命令。

关于鼠标与焦点

尽管 shell 与用户的交互全部是通过键盘来完成的，但是在终端仿真器中，也可以使用鼠标。内置到 X 窗口系统（驱动 GUI 的底层引擎）中的一种机制

可以支持快速的复制与粘贴技术。如果紧按鼠标左键选中一些文本并拖动鼠标（或双击选中一个词），该文本将复制到由 X 维护的一个缓冲区中。按下鼠标的中间按键可以将选中的文本粘贴到光标所在的位置。你可以试一下。

不要试图使用 Ctrl-C 和 Ctrl-V 在一个终端窗口内进行复制和粘贴操作，这不起作用。对于 shell 而言，这些组合键在很早之前就已经赋予了不同的含义，而那时微软的 Windows 操作系统还没有出现。

在操作上与 Windows 类似的图形桌面环境（很有可能是 KDE 或 GNOME），很可能拥有自己的焦点策略（focus policy）集合，用以通过“点击来获得焦点”。这意味着，如果一个窗口需要获得焦点（成为当前窗口），只需要点击一下即可。而传统的 X 窗口的行为是“焦点跟随着鼠标”，也就是说，当鼠标经过窗口时，窗口就会获得焦点。因此两者是截然不同的。如果没有点击窗口，那么它不会出现在前端，但此时它可以接受输入。将焦点策略设置为“焦点跟随鼠标”的方式会使终端窗口使用起来更容易。试一试吧，试过之后，你一定会喜欢上这种方式。你可在窗口管理器的配置程序中找到该设置。

1.3 几个简单的命令

在学习了键盘输入之后，我们来尝试几个简单的命令。首先是 date 命令，该命令显示当前系统的时间和日期。

```
[me@linuxbox ~]$ date
Thu Oct 25 13:51:54 EDT 2012
```

与之相关的一个命令是 cal，在默认情况下，cal 显示当月的日历。

```
[me@linuxbox ~]$ cal
    October 2012
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

如果想要查看磁盘驱动器当前的可用空间，可以使用 df 命令。

```
[me@linuxbox ~]$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda2             15115452   5012392   9949716  34% /
/dev/sda5             59631908  26545424  30008432  47% /home
/dev/sda1               147764     17370    122765  13% /boot
tmpfs                   256856         0    256856   0% /dev/shm
```

同样，要显示可用内存，可以使用 free 命令。

```
[me@linuxbox ~]$ free
          total       used       free     shared    buffers     cached
Mem:     513712     503976       9736          0       5312     122916
-/+ buffers/cache:  375748     137964
Swap:   1052248     104712     947536
```

1.4 结束终端会话

直接关闭终端窗口或是在 shell 提示符下输入 exit 命令，即可结束终端会话。

```
[me@linuxbox ~]$ exit
```

幕后的控制台

即使没有运行终端仿真器，一些终端会话也会在图形桌面的后台运行。这叫做虚拟终端或是虚拟控制台。在绝大多数系统中，通过依次按下 Ctrl-Alt-F1 键到 Ctrl-Alt-F6 组合键，可以访问大部分 Linux 发行版中的终端会话。每当访问一次会话，就会出现登录提示符，我们可以在其中输入用户名和密码。按 Alt 和 F1～F6 键，可从一个虚拟控制台转换到另一个虚拟控制台。按 Alt-F7 键可返回图形桌面环境。

第 2 章

导　航

除了在命令行进行输入操作之外，我们首先需要学习的是如何在 Linux 系统中导航文件系统。本章将介绍下述命令。

- pwd：查看当前工作目录。
- cd：改变目录。
- ls：列出目录内容。

2.1 理解文件系统树

与 Windows 相同，类 UNIX 操作系统（比如 Linux）也是以称之为分层目录结构的方式来组织文件的。这意味着文件是在树形结构的目录（有时在其他系统中称为文件夹）中进行组织的，该树形结构目录可能包含文件和其他目录。文件系统的第一个目录叫做根目录，它包含了文件和子目录。子目录里包含了更多的文件和子目录，依此类推。

需要注意的是，在 Windows 系统中，每个存储设备都有一个独立的文件系统树。而在类 UNIX 系统中，如 Linux，无论多少驱动器或存储设备与计算机相连，通常只有一个文件系统树。根据系统管理员的设置，存储设备将会连接（更准确的说是“挂载”）到文件系统树的不同位置。系统管理员要负责系统的维护。

2.2 当前工作目录

可能大部分人都熟悉用于表示文件系统树的图形文件管理器，如图 2-1 所示。需要注意的是，树通常是倒立显示的。也就是说，顶部是根目录，依次向下排列的是子目录。

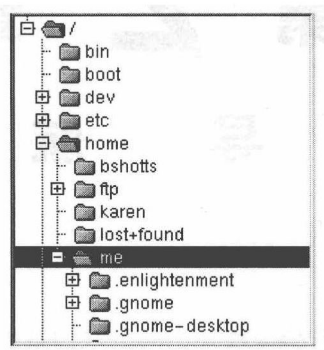


图 2-1 在图形文件管理器中显示的文件系统树

然而，由于命令行没有图像，若是要浏览文件系统树，就必须使用其他方法。

假设文件系统是一个迷宫，形如一棵倒置的树，并且用户处在文件系统之中。任何时刻，我们处在单个目录中，能够看到该目录中包含的文件、去往上一级目录（称为父目录）的路径，以及下一级的各个子目录。用户所处的目录叫做当前工作目录。使用 pwd（打印工作目录）命令可以显示当前工作目录。

```
[me@linuxbox ~]$ pwd
/home/me
```

第一次登录系统时（或是启动终端仿真器会话时），当前工作目录被设置成主目录。每个用户账号都有一个主目录，作为普通用户操作时，这是唯一一个允许用户写文件的地方。

2.3　列出目录内容

使用 ls 命令可以列出当前工作目录的文件和目录。

```
[me@linuxbox ~]$ ls
Desktop  Documents  Music  Pictures  Public  Templates  Videos
```

实际上，可以使用 ls 命令列出任何目录的内容，而不仅仅是当前工作目录。同时，它还拥有一些其他有趣的功能。我们会在第 3 章详细讨论 ls 命令。

2.4　更改当前工作目录

使用 cd 命令可以改变工作目录（即在文件系统树的位置）；只需输入 cd 命令，然后再输入目标工作目录的路径名即可。路径名指的是沿着分枝到达目标目录的路由。路径名分为两种：绝对路径名和相对路径名。首先来谈谈绝对路径名。

2.4.1　绝对路径名

绝对路径名从根目录开始，其后紧接着一个又一个文件树分支，直到到达目标目录或文件。例如，系统里有一个目录，大多数系统程序都安装到这个目录里，该目录的路径名是/usr/bin。这就意味着根目录（在路径名中用前导斜杠来表示）中有一个目录是 usr，该目录包含一个 bin 目录。

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
[me@linuxbox bin]$ ls

...Listing of many, many files ...
```

可以看到，我们已经将当前工作目录改变成/usr/bin，bin 目录中包含很多文件。请注意 shell 提示符是如何变化的。为方便起见，工作目录名通常被设置成自动显示。

2.4.2　相对路径名

绝对路径名是从根目录开始，通向目标目录，而相对路径名则是从工作目录开始的。为了实现这个目的，它通常使用一些特殊符号来表示文件系统树中的相对位置，这些特殊符号是“.”（点）和“..”（点点）。

符号“.”代表工作目录，符号“..”代表工作目录的父目录。下面演示它们是如何工作的。让我们再次将工作目录改变成/usr/bin。

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

好的，下面来说明一下，我们希望将工作目录改变成/usr/bin 的父目录，即/usr。有两种方法可以实现，一种是使用绝对路径名。

```
[me@linuxbox bin]$ cd /usr
[me@linuxbox usr]$ pwd
/usr
```

另一种是使用相对路径名。

```
[me@linuxbox bin]$ cd ..
[me@linuxbox usr]$ pwd
/usr
```

由于两种不同的方法产生同样的结果。那么我们究竟应该用哪一种方法呢？那就选择输入字符最少的吧。

同样，可以用两种方法将工作目录从/usr 变到/usr/bin。我们可以使用绝对路径名。

```
[me@linuxbox usr]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

我们也可以使用相对路径名。

```
[me@linuxbox usr]$ cd ./bin
[me@linuxbox bin]$ pwd
/usr/bin
```

必须在这里指出来的是，几乎在所有的情况下都可以省略“./”，因为它是隐含的。输入以下代码。

```
[me@linuxbox usr]$ cd bin
```

该代码与使用相对路径名的代码具有相同效果。一般而言，如果没有指定路径名，则默认为工作目录。

2.4.3 一些有用的快捷方式

表 2-1 列出了一些可以快速改变当前工作目录的方法。

表 2-1　cd 快捷方式

快捷方式	结果
cd	将工作目录改变成主目录
cd-	将工作目录改变成先前的工作目录
cd~*username*	将工作目录改变为 username 的主目录。例如，cd~bob 将目录改变成用户 bob 的主目录

有关文件名的一些重要说明

- 以“.”字符开头的文件名是隐藏的。这仅说明 ls 不会列出这些文件，除非输入 ls-a。在创建用户账号时，主目录里会放置一些隐藏文件，用来配置账号信息。仔细观察这样的文件，可以使我们了解如何自定义工作环境。此外，一些应用程序也会将它们的配置文件和设置文件以隐藏文件的形式放在主目录下面。
- 与 UNIX 一样，在 Linux 中，文件名与命令是区分大小写的。文件名 File1 和 file1 指向不同的文件。
- 与其他一些操作系统一样，Linux 没有文件扩展名的概念。我们可以按照自己的喜好随意给文件命名。文件的内容或用途由其他方式来决定。尽管类 UNIX 操作系统不使用文件扩展名来决定文件内容或用途，但是一些应用程序却这么做了。
- Linux 支持长文件名，文件名可能包含了嵌入的空格和标点符号。但是在创建文件名的过程中，仅句号、连字符和下划线是可以使用的。更为重要的是，文件名中不要嵌入空格。文件名中嵌入空格会使很多命令行任务的实现变得困难，在第 7 章中我们就会发现这个问题。如果想要表示文件名词语间的空格，可以使用下划线，以后我们就会知道这样的好处了。

第 3 章
Linux 系统

既然已经知道了如何在文件系统中跳转，是时候开始 Linux 操作系统之旅了。但是，在开始之前，我们要先学习一些对研究 Linux 系统很有帮助的命令。

- ls：列出目录内容。
- file：确定文件类型。
- less：查看文件内容。

3.1 ls 命令的乐趣

有充分的理由证明，ls 命令很可能是用户最常使用的命令。通过 ls 命令可以查看目录内容，确定各种重要文件和目录的属性。我们已经看到，只需输入 ls 命令，即可查看当前工作目录中包含的一系列文件和子目录。

```
[me@linuxbox ~]$ ls
Desktop  Documents  Music  Pictures  Public  Templates  Videos
```

除了当前工作目录之外，我们还可以指定要显示的目录，如下所示。

```
me@linuxbox ~]$ ls /usr
bin   games   kerberos   libexec   sbin   src
etc   include lib        local     share  tmp
```

我们甚至可以指定多个目录。下面这个例子就列出了用户主目录（由符号“～”表示）和/usr 目录的内容。

```
[me@linuxbox ~]$ ls ~ /usr
/home/me:
Desktop  Documents  Music  Pictures  Public  Templates  Videos
/usr:
bin   games   kerberos  libexec  sbin   src
etc   include lib       local    share  tmp
```

我们也可以改变输出格式来得到更多细节。

```
[me@linuxbox ~]$ ls -l
total 56
drwxrwxr-x 2 me me 4096 2012-10-26 17:20 Desktop
drwxrwxr-x 2 me me 4096 2012-10-26 17:20 Documents
drwxrwxr-x 2 me me 4096 2012-10-26 17:20 Music
drwxrwxr-x 2 me me 4096 2012-10-26 17:20 Pictures
drwxrwxr-x 2 me me 4096 2012-10-26 17:20 Public
drwxrwxr-x 2 me me 4096 2012-10-26 17:20 Templates
drwxrwxr-x 2 me me 4096 2012-10-26 17:20 Videos
```

在命令中加上-l，我们可以将输出以长格式显示。

3.1.1 选项和参数

下面，让我们来了解一下大部分命令是如何工作的，这也是非常重要的一点。通常，命令后面跟有一个或多个选项，带有不同选项的命令其功能也不一样。此外，命令后面还会跟有一个或多个参数，这些参数是命令作用的对象。所以大部分命令看起来如下所示：

```
command -options arguments
```

大部分命令使用的选项是在单个字符前加上连字符，如-l。但是，很多命令，包括 GNU 项目里的命令，也支持在单字前加两个连字符的长选项。而且，很多命令也允许多个短选项串在一起使用。在下面的例子中，ls 命令包含了两个选项：l 选项产生长格式输出，而 t 选项则表示以文件修改时间的先后将结果进行排序。

```
[me@linuxbox ~]$ ls -lt
```

加上长选项--reverse，则结果会以相反的顺序输出：

```
[me@linuxbox ~]$ ls -lt --reverse
```

ls 命令有大量可用的选项。最常用的选项如表 3-1 所示。

表 3-1 ls 命令的常用选项

选项	长选项	含义
-a	--all	列出所有文件，包括以点号开头的文件，这些文件通常是不列出来的（比如隐藏的文件）
-d	--directory	通常，如果指定了一个目录，ls 命令会列出目录中的内容而不是目录本身。将此选项与-l 选项结合使用，可查看目录的详细信息，而不是目录中的内容
-F	--classify	选项会在每个所列出的名字后面加上类型指示符（例如，如果名字是目录名，则会加上一个斜杠）
-h	--human-readable	以长格式列出，以人们可读的方式而不是字节数来显示文件大小
-l		使用长格式显示结果
-r	--reverse	以相反的顺序显示结果。通常，ls 命令按照字母升序排列显示结果
-S		按文件大小对结果排序
-t		按修改时间排序

3.1.2 进一步了解长列表格式

前面看到，-l 选项使得 ls 命令以长格式显示其结果。这种格式包含了大量的有用信息。下面的例子来自 Ubuntu 系统。

```
-rw-r--r-- 1 root root 3576296 2012-04-03 11:05 Experience ubuntu.ogg
-rw-r--r-- 1 root root 1186219 2012-04-03 11:05 kubuntu-leaflet.png
-rw-r--r-- 1 root root   47584 2012-04-03 11:05 logo-Edubuntu.png
-rw-r--r-- 1 root root   44355 2012-04-03 11:05 logo-Kubuntu.png
-rw-r--r-- 1 root root   34391 2012-04-03 11:05 logo-Ubuntu.png
-rw-r--r-- 1 root root   32059 2012-04-03 11:05 oo-cd-cover.odf
-rw-r--r-- 1 root root  159744 2012-04-03 11:05 oo-derivatives.doc
-rw-r--r-- 1 root root   27837 2012-04-03 11:05 oo-maxwell.odt
-rw-r--r-- 1 root root   98816 2012-04-03 11:05 oo-trig.xls
-rw-r--r-- 1 root root  453764 2012-04-03 11:05 oo-welcome.odt
-rw-r--r-- 1 root root  358374 2012-04-03 11:05 ubuntu Sax.ogg
```

再来看一下其中一个文件的不同字段，表 3-2 列出了这些不同字段的含义。

表 3-2 ls 长列表字段

字段	含义
-rw-r-r--	对文件的访问权限。第一个字符表示文件的类型。在不同类型之间，开头的“-”表示该文件是一个普通文件，d 表示目录。紧接着的三个字符表示文件所有者的访问权限，再接着的三个字符表示文件所属组中成员的访问权限，最后三个字符表示其他所有人的访问权限。详细解释请见第 9 章

续表

字段	含义
1	文件硬链接数目。链接的内容将在本章后面讨论
root	文件所有者的用户名
root	文件所属用户组的名称
32059	以字节数表示的文件大小
2012-04-03 11:05	上次修改文件的日期和时间
oo-cd-cover.odf	文件名

3.2 使用 file 命令确定文件类型

在我们探索系统的过程中，知道文件包含的内容是非常有用的。为此，我们可以使用 file 命令来确定文件类型。先前讲过，Linux 系统中的文件名不需要反映文件的内容。例如，当我们看到 picture.jpg 这样一个文件名，会很自然地觉得该文件中包含一张 JPEG 压缩图像，但是在 Linux 中却没有这个必要。我们可以这样调用 file 命令：

```
file filename
```

调用后，file 命令会打印出文件内容的简短说明。例如：

```
[me@linuxbox ~]$ file picture.jpg
picture.jpg: JPEG image data, JFIF standard 1.01
```

文件的种类有很多。事实上，在类 UNIX 操作系统（比如 Linux 系统）中，有个普遍的观念是“所有的东西都是一个文件”。随着课程的深入，大家会明白这句话的真谛。

尽管我们已经很熟悉系统中的许多文件，比如 MP3 和 JPEG 文件。但是也有一些文件比较含蓄，还有一些文件对我们而言相当陌生。

3.3 使用 less 命令查看文件内容

less 命令是一种查看文本文件的程序。纵观 Linux 系统，很多文件都含有人们可以阅读的文本。less 程序为我们查看文件提供了方便。

为什么要查看文本文件呢？因为包含系统设置的多数文件（即配置文件）是以这种格式存储的，阅读这些文件有利于更好地理解系统是如何工作的。此外，系统使用的许多实际程序（称之为脚本）也是以这种格式存储的。在随后的章节中，我们将学习如何编写文本文件来修改系统设置，并编写自己的脚本，

而现在我们只需看看它们的内容即可。

什么是文本

有很多方式可在计算机中表达信息。所有的方式都涉及在信息与一些数字之间确立一种关系，而这些数字可以用来表达信息。毕竟，计算机只能理解数字，并且所有的数据都将转换成数值来表示。

有些表示方法非常复杂（例如压缩后的视频文件），也有一些其他方法相当简单。其中出现最早也是最简单的是 ASCII 文本。ASCII（发音是“As-Key”）是美国信息交换标准码的简称。这个简单的编码方案最早用于电传打字机。

文本是字符与数字之间简单的一对一映射，它非常紧凑。由 50 个字符构成的文本在转换为数据时，也是 50 个字节。这与文字处理器文档中的文本是不一样的，比如由 Microsoft Word 或 OpenOffice.org Write 创建的文档。与简单的 ASCII 文本相比，那些文件包含了很多用来描述结构和格式的非文本元素。而普通的 ASCII 文本文件仅包含字符本身和一些基本的控制代码，比如制表符、回车符和换行符。

纵观 Linux 系统，很多文件是以文本格式存储的，并且也有很多 Linux 工具来处理文本文件。甚至连 Windows 系统也认识到这种格式的重要性。众所周知的记事本程序就是一款普通的 ASCII 文本文件编辑器。

less 命令的使用方式如下。

```
less filename
```

一旦运行起来，less 程序允许我们前后滚动文件。例如，想要查看定义了系统用户账户的文件，可输入下面的命令。

```
[me@linuxbox ~]$ less /etc/passwd
```

一旦 less 程序运行起来，我们就可查看文件内容。如果文件不止一页，可以上下滚动文件。按 Q 键可退出 less 程序。

表 3-3 列出了 less 程序最常使用的键盘命令。

表 3-3　less 命令

命令	功能
PAGE UP 或 b	后翻一页
PAGE DOWN 或 Spacebar	前翻一页
向上箭头键	向上一行

续表

命令	功能
向下箭头键	向下一行
G	跳转到文本文件的末尾
1G 或 g	跳转到文本文件的开头
/*charecters*	向前查找指定的字符串
n	向前查找下一个出现的字符串，这个字符串是之前所指定查找的
h	显示帮助屏幕
q	退出 less

少即是多（LESS IS MORE）

设计 less 程序是为了替换早期 UNIX 中的 more 程序。less 这个名字是对短语“less ls more”开了个玩笑，该短语是现代派建筑师和设计师们的座右铭。

less 命令属于“页面调度器（pagers）”程序类，这些程序允许通过一页一页的方式，轻松浏览很长的文本文档。而 more 程序只允许向前翻页，使用 less 命令既可以前后翻页，还具有很多其他的特性。

3.4 快速浏览

在 Linux 系统中，文件系统布局与其他类 UNIX 系统很相似。实际上，一个已经发布的名为 Linux 文件系统层次标准（Linux Filesystem Hierarchy Standard）的标准，已经详细阐述了这个设计。并不是所有 Linux 发行版都严格符合该标准，但大部分与之很接近。

接下来，我们将通过对文件系统的探索来找到 Linux 系统正常运行所依赖的基础。这将提供一个练习导航技巧的机会。此时我们会发现，很多有趣的文件都是普通的可读文本。请尝试下面的步骤。

1. 使用 cd 命令进入一个给定的目录。
2. 使用 ls-l 命令列出目录内容。
3. 如果看到一个感兴趣的文件，使用 file 命令确定文件内容。
4. 如果文件看起来像是一个文本，试着使用 less 命令浏览其内容。

注意　牢记复制与粘贴技巧！如果使用鼠标的话，可以双击文件名来复制，中键单击将其粘贴进命令行。

当我们浏览文件系统时，不要担心将文件系统的布局弄得混乱不堪。普通用户不具有管理文件系统的权限，那是系统管理员的工作！如果一条命令无法执行某些功能，那么继续选择其他命令。多花些时间浏览一下。系统是需要大家探索的。记住，Linux 没有秘密可言。

表 3-4 只列出了一些探索到的目录，剩余的目录请大家自由地探索！

表 3-4 在 Linux 系统中找到的目录

目录	内容
/	根目录，一切从这里开始
/bin	包含系统启动和运行所必需的二进制文件（程序）
/boot	包含 Linux 内核、最初的 RAM 磁盘映像（系统启动时，驱动程序会用到），以及启动加载程序 有趣的文件： • /boot/grub/grub.conf 或 menu.lst，用来配置启动加载程序 • /boot/vmlinuz，Linux 内核
/dev	这是一个包含设备节点的特殊目录。“把一切当成文件”也适用于设备。内核将它能够识别的所有设备存放在这个目录里
/etc	/etc 目录包含了所有系统层面的配置文件，同时也包含了一系列 shell 脚本，系统每次启动时，这些 shell 脚本都会打开每个系统服务。该目录中包含的内容都应该是可读的文本文件。 有趣的文件：尽管/etc 目录中的任何文件都很有趣，但这里只列出了一些我一直喜欢的文件： • /etc/crontab，该文件定义了自动化任务运行的时间 • /etc/fstab，存储设备以及相关挂载点的列表 • /etc/passwd，用户账号列表
/home	在通常的配置中，每个用户都会在/home 目录中拥有一个属于自己的目录。普通用户只能在自己的主目录中创建文件。这一限制可以保护系统免遭错误的用户行为的破坏
/lib	包含核心系统程序使用的共享库文件。这与 Windows 系统中的 DLL 类似
/lost+found	每个使用 Linux 文件系统的格式化分区或设备，例如 ext3 文件系统，都会有这个目录。当文件系统崩溃时，该目录用于恢复分区。除非系统真的发生很严重的问题，否则这个目录一直是空的
/media	在现代 Linux 系统中，/media 目录包含可移除媒体设备的挂载点。比如 USB 驱动、CD-ROM 等。这些设备在插入计算机后，会自动挂载到这个目录节点下
/mnt	在早期的 Linux 系统中，/mnt 目录包含手动挂载的可移除设备的挂接点
/opt	/opt 目录用来安装其他可选的软件。主要用来存放可能安装在系统中的商业软件
/proc	/proc 目录很特殊。从文件的角度来说，它不是存储在硬盘中的真正的文件系统，反而是一个 Linux 内核维护的虚拟文件系统。它包含的文件是内核的窥视孔。该文件是可读的，从中可以看到内核是如何监管计算机的
/root	root 账户的主目录
/sbin	该目录放置“系统”二进制文件。这些程序执行重要的系统任务，这些任务通常是为超级用户预留的

续表

目录	内容
/tmp	/tmp 是供用户存放各类程序创建的临时文件的目录。某些配置使得每次系统重启时都会清空该目录
/usr	/usr 目录可能是 Linux 系统中最大的目录树。它包含普通用户使用的所有程序和相关文件
/usr/bin	/usr/bin 目录中放置了一些 Linux 发行版安装的可执行程序。该目录通常会存储成千上万个程序
/usr/lib	/usr/bin 目录中的程序使用的共享库
/usr/local	这个/usr/local 目录是并非系统发行版自带，但却打算让系统使用的程序的安装目录。由源代码编译好的程序通常安装在/usr/local/bin 中。在一个新安装的 Linux 系统中，就存在这一个目录，但却是空目录，直到系统管理员向其中添加内容
/usr/sbin	包含更多的系统管理程序
/usr/share	/usr/share 目录里包含了/usr/bin 中的程序所使用的全部共享数据，这包括默认配置文件、图标、屏幕背景、音频文件等
/usr/share/doc	安装在系统中的大部分程序包包含一些文档文件。在/usr/share/doc 中，文档文件是按照软件包来组织分类的
/var	除了/tmp 和/home 目录之外，目前看到的目录相对来说都是静态的；也就是说，其包含的内容是不变的。而那些可能改变的数据存储在/var 目录树里。各种数据库、假脱机文件、用户邮件等都存储在这里
/var/log	/var/log 目录包含的日志文件，记录了各种系统活动。这些文件非常重要，并且应该时不时地监控它们。其中最有用的文件是/var/log/messages。注意，为了安全起见，在一些系统里，必须是超级用户才能查看日志文件

3.5 符号链接

在浏览过程中，我们可能会看到带有如下条目的目录信息。

```
lrwxrwxrwx 1 root root   11 2012-08-11  07:34 libc.so.6 -> libc-2.6.so
```

注意，该条目信息的第一个字母是 l，而且看起来像是有两个文件名。这种特殊的文件叫做符号链接（又叫软链接或 symlink）。在大多类 UNIX 系统中，一个文件很可能采用多个名字来引用。虽然这种特性的意义并不明显，但是它真的很有用。

想象这样一个场景：一个程序需要使用包含在 foo 文件中的一个共享资源，但 foo 版本变化很频繁。这样，在文件名中包含版本号会是一个好主意，因此管理员或其他相关方就能够看到安装了 foo 的哪个版本。这就出现一个问题。如果改变了共享资源的名称，就必须跟踪每个可能使用了该共享资源的程序，

并且当安装了该资源新的版本后，都要让使用它的程序去寻找新的资源名。这听起来很没有意思。

下面就是符号链接的用武之处。假设 foo 的安装版本是 2.6，它的文件名是 foo-2.6，然后创建一个符号链接 foo 指向 foo-2.6。这意味着，当一个程序打开 foo 文件时，它实际上打开的是文件 foo-2.6，这样一来皆大欢喜。依赖 foo 文件的程序能够找到它，并且也能看到实际安装的版本。当需要升级到 foo-2.7 时，只需将该文件添加到系统里，删除符号链接文件 foo，创建一个指向新版本的符号链接即可。这不仅解决了版本升级的问题，也可以将两种版本都保存在机器里。假如 foo-2.7 存在一个程序错误（都怪该死的开发商！），需要切换到旧的版本。同样，只需删除指向新版本的符号链接，重新创建指向旧版本的符号链接即可。

以上列举的目录（来源于 Fedora 系统的/lib 目录）中显示了一个指向 libc-2.6.so 共享库文件的符号链接 libc.so.6。也就是说，搜索文件 libc.so.6 的程序实际上访问的是 libc-2.6.so 文件。下一章我们将学习如何创建符号链接。

硬链接

既然在讨论链接问题，就需要提一下另一种类型的链接，即硬链接。它同样允许文件有多个名字，但是处理的方式是不同的。下一章我们将进一步讨论符号链接与硬链接的区别。

第 4 章

操作文件与目录

现在，我们准备好做些实际工作了！本章将介绍如下命令。

- cp：复制文件和目录。
- mv：移动或重命名文件和目录。
- mkdir：创建目录。
- rm：移除文件和目录。
- in：创建硬链接和符号链接。

这 5 个命令属于最常使用的 Linux 命令之列，可用来操作文件与目录。

坦率地说，使用图形文件管理器来执行一些由这些命令执行的任务要容易得多。使用文件管理器，我们可以将文件从一个目录拖放到另一个目录，我们可以剪切和粘贴文件，我们可以删除文件。那么，为什么还要用这些命令行程序呢？

原因就在于命令行程序具有强大的功能和灵活的操作。虽然使用图形文件

管理器能轻松实现简单的文件操作，但是对于复杂的任务，使用命令行程序更容易完成。例如，怎样仅因为文件在目标目录中不存在或存在旧的版本，就将所有 HTML 文件从一个目录复制到目标目录里呢？要完成这个任务，使用文件管理器则相当困难，而使用命令行则很容易。

```
cp -u *.html destination
```

4.1 通配符

在开始使用命令之前，我们需要介绍一个使命令行如此强大的 shell 特性。由于 shell 需要经常使用文件名，因此它提供了一些特殊字符来帮助你快速指定一组文件名。这些特殊字符称为通配符。通配符（也叫文件名替换）允许用户依据字符模式选择文件名。表 4-1 列出了通配符以及它们所选择的对象。

表 4-1　通配符

通配符	匹配项
*	匹配任意多个字符（包括 0 个和 1 个）
?	匹配任一单个字符（不包括 0 个）
[characters]	匹配任意一个属于字符集中的字符
[!characters]	匹配任意一个不属于字符集中的字符
[[:class:]]	匹配任意一个属于指定字符类中的字符

表 4-2 列出了最常用的字符类。

表 4-2　常用的字符类

字符类	匹配项
[:alnum:]	匹配任意一个字母或数字
[:alpha:]	匹配任意一个字母
[:digit]	匹配任意一个数字
[:lower:]	匹配任意一个小写字母
[:upper:]	匹配任意一个大写字母

通配符的使用使得为文件名构建复杂的筛选标准成为可能。表 4-3 列出了一些通配符模式及其匹配内容的示例。

表 4-3　通配符示例

模式	匹配项
*	所有文件

续表

形式	匹配项
g*	以 g 开头的任一文件
b*.txt	以 b 开头，中间有任意多个字符，并以.txt 结尾的任一文件
Data???	以 Data 开头，后面跟 3 个字符的任一文件
[abc]*	以 abc 中的任一个开头的任一文件
BACKUP.[0-9][0-9][0-9]	以 BACKUP.开头，后面紧跟 3 个数字的任一文件
[[:upper:]]*	以大写字母开头的任一文件
[![:digit:]]*	不以数字开头的任一文件
*[[:lower:]123]	以小写字母或数字 1、2、3 中的任一个结尾的任一文件

通配符可以与任一个使用文件名为参数的命令一起使用，在第 7 章我们会进一步讨论。

字符范围

如果你之前使用过其他的类 UNIX 环境或是读过该主题的相关书籍，可能遇到过[A-Z]或[a-z]形式的字符范围表示法。这些都是传统的 UNIX 表示法，在早期的 Linux 版本中仍然奏效。尽管它们仍然起作用，但使用时请务必小心，因为一旦配置不当，就会产生非预期的结果。目前，我们要避免使用它们，而是使用字符类。

通配符在 GUI 中也奏效

通配符相当有用，不仅仅因为它们在命令行中使用频繁，而且在于一些图形文件管理器也支持通配符操作。

- 在 Nautilus（GNOME 的文件管理器）中，你可以使用 Edit->Select Pattern 选择文件。仅仅输入一个用通配符表示的文件选择模式后，则当前查看的目录中，所匹配的文件就会突出显示。
- 在 Dolphin 和 Konqueror（KDE 的文件管理器）的一些版本中，你可以直接在地址栏输入通配符。比如，如果想要在/usr/bin 目录中查找所有以小写字符 u 开始的文件，只需在地址栏输入/usr/bin/u*，即可显示匹配的结果。

最初源于命令行界面的许多理念，也同样适用于图形界面。而这恰恰是使 Linux 系统桌面如此强大的原因之一。

4.2 mkdir——创建目录

mkdir 命令是用来创建目录的，格式如下。

```
mkdir directory...
```

注意表示法：本书在描述命令时，如果参数后面带有 3 个点号（如上所示），这表示该参数重复。因此，下面这种情况：

```
mkdir dir1
```

可创建单个 dir1 目录，而输入：

```
mkdir dir1 dir2 dir3
```

可创建 3 个目录，分别命名为 dir1、dir2 和 dir3。

4.3 cp——复制文件和目录

cp 命令用来复制文件和目录。它有两种不同的使用方式，如下所示。

```
cp item1 item2
```

将单个文件或目录 item1 复制到文件或目录 item2 中。

```
cp item... directory
```

将多个项目（文件或目录）复制进一个目录中。

表 4-4 和表 4-5 列出了 cp 常用的选项（短选项和等价的长选项）。

表 4-4 cp 命令选项

选项	含义
-a，--archive	复制文件和目录及其属性，包括所有权和权限。通常来说，复制的文件具有用户所操作文件的默认属性
-i，--interactive	在覆盖一个已存在的文件前，提示用户进行确认。如果没有指定该选项，cp 会默认覆盖文件
-r，--recursive	递归地复制目录及其内容。复制目录时需要这个选项（或-a 选项）
-u，--update	当将文件从一个目录复制到另一个目录时，只会复制那些目标目录中不存在的文件或是目标目录相应文件的更新文件
-v，--verbose	复制文件时，显示信息性消息（informative message）

表 4-5 cp 命令示例

命令	结果
cp file1 file2	将 file1 复制到 file2。如果 file2 存在，则会被 file1 的内容覆盖。如果 file2 不存在，则创建 file2
cp –i file1 file2	同上，区别在于当 file2 存在时，覆盖之前通知用户确认
cp file1 file2 dir1	将 file1 和 file2 复制到目录 dir1 里。dir1 必须已经存在
cp dir1/* dir2	通过使用通配符，将 dir1 中的所有文件复制的 dir2 中。dir2 必须已经存在
cp –r dir1 dir2	将 dir1 目录（及其内容）复制到 dir2 目录中。如果 dir2 不存在，创建 dir2，且包含与 dir1 目录相同的内容

4.4 mv——移除和重命名文件

mv 命令可以执行文件移动和文件重命名操作，这具体取决于如何使用它。在这两种情况下，完成操作之后，原来的文件名将不存在。mv 的使用方法与 cp 基本相似。

`mv item1 item2`

将文件（或目录）item1 移动（或重命名）为 item2，或是

`mv item... directory`

将一个或多个条目从一个目录移动到另一个目录下。

mv 命令很多选项与 cp 命令是共享的，如表 4-6 和表 4-7 所示。

表 4-6 mv 选项

选项	含义
-i，--interactive	覆盖一个已存在的文件之前，提示用户确认。如果没有指定该选项，mv 会默认覆盖文件
-u，--update	将文件从一个目录移动到另一个目录，只移动那些目标目录中不存在的文件或是目标目录里相应文件的更新文件
-v，--verbose	移动文件时显示信息性消息

表 4-7 mv 示例

命令	结果
mv file1 file2	将 file1 移到 file2。如果 file2 存在，则会被 file1 的内容覆盖。如果 file2 不存在，则创建 file2。无论哪一种情况，file1 不再存在
mv –i file1 file2	同上，仅当 file2 存在时，覆盖之前通知用户确认
mv file1 file2 dir1	将 file1 和 file2 移到目录 dir1 下。dir1 必须已经存在
mv dir1 dir2	将目录 dir1（和其内容）移到目录 dir2 下。如果目录 dir2 不存在，创建目录 dir2，将目录 dir1 的内容移到 dir2 下，同时删除目录 dir1

4.5 rm——删除文件和目录

rm 命令用来移除（删除）文件和目录，如下所示。

```
rm item…
```

其中，item 是一个或多个文件（或目录）的名称。

小心 rm 命令

类 UNIX 操作系统（如 Linux）并不包含还原删除操作的命令。一旦使用 rm 命令，就彻底地删除了。Linux 系统默认用户是明智的，并清楚自己在干什么。

rm 命令与通配符在一起使用时要特别小心。来看下面这个经典的示例。假设我们只希望删除目录中的 HTML 文件，为此需要输入以下正确的命令：

```
rm *.html
```

如果不小心在*与.html 之间多打了一个空格，如下所示：

```
rm * .html
```

rm 命令将会删除目录中所有文件，并提示说明目录中没有叫做.html 的文件。

提示：当 rm 命令与通配符一起使用时，除仔细检查输入内容外，可使用 ls 命令预先对通配符做出测试，这将显示欲删除的文件。紧接着，你可以按下向上方向键调用之前的命令，并使用 rm 代替 ls。

表 4-8 和表 4-9 列出了 rm 命令的一些常用选项。

表 4-8 rm 选项

选项	含义
-i，--interactive	删除一个已存在的文件前，提示用户确认。如果没有指定这个选项，rm 命令会默认删除文件
-r，--recursive	递归地删除目录。也就是说，如果删除的目录有子目录的话，也要将其删除。要删除一个目录，则必须指定该选项
-f，--force	忽略不存在的文件并无需提示确认。该选项会覆盖--interactive 选项
-v，--verbose	删除文件时显示信息性消息

表 4-9　rm 实例

命令	结果
rm file1	在不提示用户的情况下，删除 file1
rm –i file1	删除 file1 前，提示用户确认
rm –r file1 dir1	删除 file1、dir1 以及它们的内容
rm –rf file1 dir1	同上，当时在 file1 或 dir1 不存在时，rm 仍会继续执行，且不提示用户

4.6 ln——创建链接

ln 命令可用来创建硬链接或是符号链接。它的使用方式有两种。

```
ln file link
```

用来创建硬链接，而

```
ln –s item link
```

用来创建符号链接，这里的 item 可以是文件也可以是目录。

4.6.1 硬链接

硬链接是最初 UNIX 用来创建链接的方式，符号链接较之更为先进。默认情况下，每个文件有一个硬链接，该硬链接会给文件起名字。当创建一个硬链接的时候，也为这个文件创建了一个额外的目录条目。硬链接有两条重要的局限性。

- 硬链接不能引用自身文件系统之外的文件。也就是说，链接不能引用与该链接不在同一磁盘分区的文件。
- 硬链接无法引用目录。

硬链接和文件本身没有什么区别。与包含符号链接的目录列表不同，包含硬链接的目录列表没有特别的链接指示说明。当硬链接被删除时，只是删除了这个链接，但是文件本身的内容依然存在（也就是说，该空间没有释放），除非该文件的所有链接都被删除了。

因为会经常遇到它们，了解硬链接就显得特别重要。但是现在大多使用的是符号链接，下面将会有所介绍。

4.6.2 符号链接

符号链接是为了克服硬链接的局限性而创建的。符号链接是通过创建一个特殊类型的文件来起作用的，该文件包含了指向引用文件或目录的文本指针。就这点来看，符号链接与 Windows 系统下的快捷方式非常相似，但是，符号链接要早于 Windows 的快捷方式很多年。

符号链接指向的文件与符号链接自身几乎没有区别。例如，将一些东西写进符号链接里，那么这些东西同样也写进了引用文件。而当删除一个符号链接时，删除的只是符号链接而没有删除文件本身。如果先于符号链接之前删除文件，那么这个链接依然存在，但却不指向任何文件。此时，这个链接就称为坏链接。在很多实现中，ls 命令会用不同的颜色来显示坏链接，比如红色，从而显示它们的存在。

链接的概念似乎很令人迷惑，但是不要害怕。我们要经常性地使用，它们慢慢的就会清晰起来。

4.7 实战演练

由于我们要做一些实际的文件操作，我们先来创建一个安全的地带，来执行文件操作命令。首先，我们需要一个工作目录。我们在主目录里创建一个目录并命名为 playground。

4.7.1 创建目录

mkdir 命令用来创建一个目录。为了创建 playground 目录，我们首先要保证当前目录是主目录，然后再创建新目录。

```
[me@linuxbox ~]$ cd
[me@linuxbox ~]$ mkdir playground
```

为了让我们的实战演练更有意思，我们在 playground 目录中新建两个目录，命名为 dir1、dir2。为此，我们应先将当前的工作目录切换为 playground，然后再次执行 mkdir 命令。

```
[me@linuxbox ~]$ cd playground
[me@linuxbox playground]$ mkdir dir1 dir2
```

需要注意的是，mkdir 命令可以接受多个参数，从而允许我们用一个命令创建两个目录。

4.7.2 复制文件

接下来，让我们在创建的目录中放入一些数据。这一过程可通过文件复制来完成。通过使用 cp 命令，我们可以将/ect 目录中的 passwd 文件复制到当前工作目录里。

```
[me@linuxbox playground]$ cp /etc/passwd .
```

注意我们是如何使用当前工作目录的快捷方式的，即在命令末尾加单个句点。如果我们此时执行 ls 命令，将会看到我们的文件。

```
[me@linuxbox playground]$ ls -l
total 12
drwxrwxr-x 2 me me 4096 2012-01-10 16:40 dir1
drwxrwxr-x 2 me me 4096 2012-01-10 16:40 dir2
-rw-r--r-- 1 me me 1650 2012-01-10 16:07 passwd
```

现在让我们使用-v 选项，重复操作复制命令，来看看结果如何。

```
[me@linuxbox playground]$ cp -v /etc/passwd .
'/etc/passwd' -> './passwd'
```

cp 命令再次执行复制操作，但是，这一次显示了一条简洁的信息来指明它正在执行什么操作。需要注意的是，在没有任何警告的情况下，cp 命令覆盖了第一次的复制内容。cp 命令会假定用户清楚自己当前的操作。加上-i（交互式）选项可以获得警告信息。

```
[me@linuxbox playground]$ cp -i /etc/passwd .
cp: overwrite './passwd'?
```

通过在提示符下输入 y，文件就会被重写；任何其他的字符（比如，n）会使 cp 命令保留该文件。

4.7.3 移动和重命名文件

现在，passwd 这个名字似乎没有那么有趣，而我们毕竟是在进行实战演练，因此我们给它改个名字。

```
[me@linuxbox playground]$ mv passwd fun
```

现在传送 fun 文件，这是通过将重命名的文件移动到各个目录，然后再移动回当前目录来实现的：

```
[me@linuxbox playground]$ mv fun dir1
```

首先移到目录 dir1 下，然后：

```
[me@linuxbox playground]$ mv dir1/fun dir2
```

将文件从目录 dir1 移到 dir2，然后：

```
[me@linuxbox playground]$ mv dir2/fun .
```

再将文件 fun 重新移到当前工作目录下。下面来看 mv 命令的效果。首先，再次将数据文件移到目录 dir1。

```
[me@linuxbox playground]$ mv fun dir1
```

然后将目录 dir1 移到 dir2 并且使用 ls 命令进行确认。

```
[me@linuxbox playground]$ mv dir1 dir2
[me@linuxbox playground]$ ls -l dir2
total 4
drwxrwxr-x 2 me me 4096 2012-01-11 06:06 dir1
[me@linuxbox playground]$ ls -l dir2/dir1
total 4
-rw-r--r-- 1 me me 1650 2012-01-10 16:33 fun
```

注意，因为目录 dir2 已经存在，mv 命令将目录 dir1 移到 dir2。如果 dir2 不存在，mv 将 dir1 重命名为 dir2。最后，我们将所有东西放回原处。

```
[me@linuxbox playground]$ mv dir2/dir1 .
[me@linuxbox playground]$ mv dir1/fun .
```

4.7.4 创建硬链接

现在，我们试着创建一些链接。首先是创建硬链接，我们先按照如下方式创建一些指向数据文件的链接：

```
[me@linuxbox playground]$ ln fun fun-hard
[me@linuxbox playground]$ ln fun dir1/fun-hard
[me@linuxbox playground]$ ln fun dir2/fun-hard
```

目前有 4 个文件 fun 的实例。来看一下 playground 目录。

```
[me@linuxbox playground]$ ls -l
total 16
drwxrwxr-x 2 me  me  4096 2012-01-14 16:17 dir1
drwxrwxr-x 2 me  me  4096 2012-01-14 16:17 dir2
-rw-r--r-- 4 me  me  1650 2012-01-10 16:33 fun
-rw-r--r-- 4 me  me  1650 2012-01-10 16:33 fun-hard
```

可以注意到，在列表中，文件 fun 和 fun-hard 的第二个字段都是 4，这是文件 fun 存在的硬链接数目。你要记得，由于文件名是由链接创建的，所以一个文件通常至少有一个链接。那么，我们是如何知道 fun 和 fun-hard 实际上是同一个文件的呢？这种情况下，ls 命令无济于事。虽然从第 5 个字段得知 fun 和

fun-hard 文件大小相同，但是我们的列表并没有提供可靠的方式来确认它们是否是同一个文件。要解决这个问题，必须做进一步研究。

提到硬链接时，可以想象文件是由两部分组成的，即包含文件内容的数据部分和包含文件名的名称部分。创建硬链接时，实际上是创建了额外的名称，这些名称都指向同一数据部分。系统分配了一系列的盘块给所谓的索引节点（inode），该节点随后与文件名称部分建立关联。因此，每个硬链接都指向包含文件内容的具体索引节点。

ls 命令有一种方式可以显示上述信息。它是通过在命令中加上-i 选项来实现的。

```
[me@linuxbox playground]$ ls -li
total 16
12353539 drwxrwxr-x 2 me  me  4096 2012-01-14 16:17 dir1
12353540 drwxrwxr-x 2 me  me  4096 2012-01-14 16:17 dir2
12353538 -rw-r--r-- 4 me  me  1650 2012-01-10 16:33 fun
12353538 -rw-r--r-- 4 me  me  1650 2012-01-10 16:33 fun-hard
```

在这个列表中，第一个字段就是索引节点号，可以看到，fun 和 fun-hard 共享同一个索引节点号，这就证实它们是相同的文件。

4.7.5 创建符号链接

只所以创建符号链接，是为了克服硬链接的两大不足，即硬链接无法跨越物理设备，也无法引用目录，只能引用文件。符号链接是一种特殊类型的文件，它包含了指向目标文件或目录的文本指针。

创建符号链接与创建硬链接相似，如下所示。

```
[me@linuxbox playground]$ ln -s fun fun-sym
[me@linuxbox playground]$ ln -s ../fun dir1/fun-sym
[me@linuxbox playground]$ ln -s ../fun dir2/fun-sym
```

第一个命令相当直接，我们只是在 ln 命令中添加-s 选项，就可以创建符号链接而不是硬链接。但是，接下来的两个命令是干什么的呢？牢记一点，创建符号链接时，同时也创建一个文本来描述目标文件在哪里与与符号链接有关联。如果看看 ls 命令的输出就更容易明白了。

```
[me@linuxbox playground]$ ls -l dir1
total 4
-rw-r--r-- 4 me  me  1650 2012-01-10 16:33 fun-hard
lrwxrwxrwx 1 me  me     6 2012-01-15 15:17 fun-sym -> ../fun
```

在目录 dir1 中，fun-sym 的列表显示它是一个符号链接，这是通过第 1 个

字段中的首字符“l”来确认的，并且它也指“../fun”，这也是正确的。相对于 fun-sym 的实际位置，文件 fun 在它的上一级目录。还要注意到，符号链接文件的长度是 6，这是“../fun”字符串中字符的数目，而不是它所指向的文件的长度。

创建符号链接时，可以使用绝对路径名，如下所示：

```
[me@linuxbox playground]$ ln -s /home/me/playground/fun dir1/fun-sym
```

也可以使用相对路径，如前面的示例所示。因为相对路径允许包含符号链接的目录被重命名和/或移动，而且不会破坏链接，因此更可取一些。

除了普通文件之外，符号链接也可以引用目录。

```
[me@linuxbox playground]$ ln -s dir1 dir1-sym
[me@linuxbox playground]$ ls -l
total 16
drwxrwxr-x 2 me  me  4096 2012-01-15 15:17 dir1
lrwxrwxrwx 1 me  me     4 2012-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me  me  4096 2012-01-15 15:17 dir2
-rw-r--r-- 4 me  me  1650 2012-01-10 16:33 fun
-rw-r--r-- 4 me  me  1650 2012-01-10 16:33 fun-hard
lrwxrwxrwx 1 me  me     3 2012-01-15 15:15 fun-sym -> fun
```

4.7.6 移除文件和目录

前面讲到，使用 rm 命令可以删除文件和目录。那么我们就用它来清空 playground 目录。首先，我们删除目录中的一个硬链接。

```
[me@linuxbox playground]$ rm fun-hard
[me@linuxbox playground]$ ls -l
total 12
drwxrwxr-x 2 me  me  4096 2012-01-15 15:17 dir1
lrwxrwxrwx 1 me  me     4 2012-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me  me  4096 2012-01-15 15:17 dir2
-rw-r--r-- 3 me  me  1650 2012-01-10 16:33 fun
lrwxrwxrwx 1 me  me     3 2012-01-15 15:15 fun-sym -> fun
```

不出所料，文件 file-hard 被删除了，文件 fun 的链接数相应的也由 4 变成了 3，如目录列表的第二个字段所示。接下来，我们删除文件 fun，为了好玩，我们还会加上-i 选项，看看执行了哪些操作。

```
[me@linuxbox playground]$ rm -i fun
rm: remove regular file 'fun'?
```

在提示符下输入字符 y，文件就被删除了。现在看一下 ls 命令的输出。注意 fin-sym 文件发生了什么变化？由于它是一个符号链接，且指向的文件现在已经不存在，所以链接也就破坏了。

```
[me@linuxbox playground]$ ls -l
total 8
drwxrwxr-x 2 me  me  4096 2012-01-15 15:17 dir1
lrwxrwxrwx 1 me  me     4 2012-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me  me  4096 2012-01-15 15:17 dir2
lrwxrwxrwx 1 me  me     3 2012-01-15 15:15 fun-sym -> fun
```

大多数 Linux 发行版会配置 ls 命令，来显示破坏的链接。在 Fedora 系统中，破坏的链接是以闪烁的红色来显示的！受破坏的链接并不危险，但是会相当混乱麻烦。如果试图调用破坏的链接，将会看到如下情况：

```
[me@linuxbox playground]$ less fun-sym
fun-sym: No such file or directory
```

稍微清理一下。我们准备删除符号链接。

```
[me@linuxbox playground]$ rm fun-sym dir1-sym
[me@linuxbox playground]$ ls -l
total 8
drwxrwxr-x 2 me  me  4096 2012-01-15 15:17 dir1
drwxrwxr-x 2 me  me  4096 2012-01-15 15:17 dir2
```

有关符号链接，需要记住一点，即大部分文件操作是以链接目标为对象的，而非链接本身。而 rm 命令是个例外。当删除一个链接的时候，链接本身被删除，但是目标文件依旧存在。

最后，我们需要删除目录 playground。为此，我们将返回主目录，使用 rm 命令的递归选项（-r）来删除 playground 目录以及包括子目录在内的所有内容。

```
[me@linuxbox playground]$ cd
[me@linuxbox ~]$ rm -r playground
```

使用 GUI 创建符号链接

GNOME 和 KDE 中的文件管理器提供了一种自动创建符号链接的简单方法。在 GNOME 环境下，拖拽文件时同时按住 Ctrl-Shift 键将会新建链接文件，而不是执行复制（移动）操作。在 KDE 环境下，无论什么时候放下（drop）一个文件都会弹出一个小菜单，它提供了复制、移动或创建链接文件等选项。

4.8 本章结尾语

到此为止，我们已经学习了大量的基础知识，可能要花一段时间来完全消

化吸收。我们要反复地操作 playground 实例，直到完全掌握。其中，能够很好地理解基本的文件操作命令和通配符是很重要的。我们可以通过增加文件和目录来自由地拓展 playground 实例，使用通配符来指明各种操作需要的文件。刚开始的时候，链接的概念可能有些模糊，但在花些时间学习之后，我们就会发现它真的是大有裨益。

第 5 章 命令的使用

到此为止，我们已经学习了多个神秘的命令，而且每个命令带有神秘的选项和参数。在本章中，我们将进一步揭开命令的神秘面纱，甚至尝试创建自己的命令。本章中介绍的命令如下。

- type：说明如何解释命令名。
- which：显示会执行哪些可执行程序。
- man：显示命令的手册页。
- apropos：显示一系列合适的命令。
- info：显示命令的 info 条目。
- whatis：显示一条命令的简述。
- alias：创建一条命令的别名。

5.1 究竟什么是命令

一条命令不外乎以下 4 种情况。

- **可执行程序**。可执行程序就像在/usr/bin 目录里看到的所有文件一样。在该程序类别中，程序可以编译为二进制文件，比如 C、C++语言编写的程序，也可以是 shell、Perl、Python、Ruby 等脚本语言编写的程序。
- **shell 内置命令**。bash 支持许多在内部称之为 shell builtin 的内置命令。例如，cd 命令就是 shell 内置指令。
- **shell 函数**。shell 函数是合并到环境变量中的小型 shell 脚本。在后面的章节中，我们将讨论环境变量的配置以及 shell 函数的编写，但是目前我们只需知道它们的存在就好了。
- **alias 命令**。我们可以在其他命令的基础上定义自己的命令。

5.2 识别命令

能够准确地识别我们使用的命令是上述 4 种命令类型中的哪一种是很有用的。为此，Linux 提供了两个方法来识别命令类型。

5.2.1 type——显示命令的类型

type 命令是一个 shell 内置命令，可根据指定的命令名显示 shell 将要执行的命令类型。格式如下。

```
type command
```

这里的 command 是想要查看的命令名。一些实例如下所示。

```
[me@linuxbox ~]$ type type
type is a shell builtin
[me@linuxbox ~]$ type ls
ls is aliased to 'ls --color=tty'
[me@linuxbox ~]$ type cp
cp is /bin/cp
```

这里将看到 3 种不同命令的查看结果。需要注意的是，ls 命令（来自 Fedora 系统）实际上是带有--color=tty 选项的 ls 命令的别名。现在我们知道了 ls 命令的输出为什么会有颜色了。

5.2.2 which——显示可执行程序的位置

有时候，系统中可能会安装了一个可执行程序的多个版本。这种现象虽然在桌面系统中不常见，但是在大型服务器中却是很常见的。使用 which 命令可以确定一个给定可执行文件的准确位置。

```
[me@linuxbox ~]$ which ls
/bin/ls
```

which 命令只适用于可执行程序，而不适用于内置命令和命令别名（真正可执行程序的替代物）。试图在 shell 内置命令（例如，cd）中使用 which 命令时，要么没有响应，要么得到一条错误信息：

```
[me@linuxbox ~]$ which cd
/usr/bin/which: no cd in (/opt/jre1.6.0_03/bin:/usr/lib/qt-3.3/bin:/usr/kerber
os/bin:/opt/jre1.6.0_03/bin:/usr/lib/ccache:/usr/local/bin:/usr/bin:/bin:/home
/me/bin)
```

which 命令是一种是“无法找到命令”的奇特方式。

5.3 获得命令文档

了解了什么是命令后，我们可以查看每一类命令的可用文档。

5.3.1 help——获得 shell 内置命令的帮助文档

bash 为每一个 shell 内置命令提供了一个内置的帮助工具。输入 help，然后输入 shell 内置命令的名称即可使用该帮助工具。例如：

```
[me@linuxbox ~]$ help cd
cd: cd [-L|-P] [dir]
Change the current directory to DIR. The variable $HOME is the default DIR.
The variable CDPATH defines the search path for the directory containing DIR.
Alternative directory names in CDPATH are separated by a colon (:). A null
directory name is the same as the current directory, i.e. '.'. If DIR begins
with a slash (/), then CDPATH is not used. If the directory is not found, and
the shell option 'cdable_vars' is set, then try the word as a variable name.
If that variable has a value, then cd to the value of that variable. The -P
option says to use the physical directory structure instead of following
symbolic links; the -L option forces symbolic links to be followed.
```

注意表示法：出现在命令语法描述中的方括号表示一个可选的选项。竖线符号代表的是两个互斥的选项。比如上边的 cd 命令：cd [-L|-P] [dir]。

这种表示法说明，cd 命令后可能有一个-L 参数，也可能是-P 参数，甚至可以跟

参数 dir。

尽管 cd 命令的帮助文档简明而又准确，但这绝不是一个辅导教程，我们所可以看到，帮助文档中也似乎提到了很多我们还没有讨论到的内容！别担心，稍后我们会详加说明。

5.3.2 help——显示命令的使用信息

很多可执行程序都支持--help 选项，--help 选项描述了命令支持的语法和选项。例如：

```
[me@linuxbox ~]$ mkdir --help
Usage: mkdir [OPTION] DIRECTORY...
Create the DIRECTORY(ies), if they do not already exist.

  -Z, --context=CONTEXT (SELinux) set security context to CONTEXT
Mandatory arguments to long options are mandatory for short options too.
  -m, --mode=MODE     set file mode (as in chmod), not a=rwx - umask
  -p, --parents no    error if existing, make parent directories as
                      needed
  -v, --verbose       print a message for each created directory
      --help          display this help and exit
      --version       output version information and exit
Report bugs to <bug-coreutils@gnu.org>.
```

一些程序不支持--help 选项，但是我们还是要试试。这通常会产生一条错误消息，该错误消息也能揭示相同的命令使用信息。

5.3.3 man——显示程序的手册页

大多数供命令行使用的可执行文件，提供一个称之为 manual 或者是 man page 的正式文档。该文档可以用一种称为 man 的特殊分页程序来查看，用法如下。

```
man program
```

这里的 program 是需要查看的命令名称。

手册文档在格式上会有所不同，但是通常都包括标题、命令句法的摘要、命令用途的描述、命令选项列表以及每个命令选项的描述。但是，手册文档通常不包括实例，更多的是作为一个参考使用，而不是教程。例如，尝试查看 ls 命令的手册文档。

```
[me@linuxbox ~]$ man ls
```

在大多数 Linux 系统中，man 命令调用 less 命令来显示手册文档。所以，当显示手册文档时，你熟悉的所有 less 命令都能奏效。

man 命令显示的“手册文档”被分成多个部分（section），它不仅包括用户命令，也包括系统管理命令、程序接口、文件格式等。表 5-1 描述了手册文档的结构安排。

表 5-1　手册文档的组织结构

部分	内容
1	用户命令
2	内核系统调用的程序接口
3	C 库函数程序接口
4	特殊文件，如设备节点和驱动程序
5	文件格式
6	游戏和娱乐，例如屏幕保护程序
7	其他杂项
8	系统管理命令

有时候我们需要查看手册文档的具体部分，以查找我们需要的信息。当我们所查找的一个文件格式同时也是一个命令名的时候，这一点就尤为重要了。如果没有指明部分编号（section number），通常我们会获得第一次匹配的实例（它可能会出现在第一部分）。为了指明具体在哪个部分，我们可以这样使用 man 命令。

```
man section search_term
```

例如：

```
[me@linuxbox ~]$ man 5 passwd
```

该命令将会显示文件/etc/passwd 的文件格式描述手册。

5.3.4 apropos——显示合适的命令

我们有可能会搜索参考手册列表，才进行基于某个搜索条目的匹配。尽管有些粗糙，但是这种方法有时还是很有用的。下面是一个使用 floppy 为搜索条目，来搜索参考手册的例子。

```
[me@linuxbox ~]$ apropos floppy
create_floppy_devices  (8) - udev callout to create all possible
                             floppy device based on the CMOS type
fdformat               (8) - Low-level formats a floppy disk
floppy                 (8) - format floppy disks
gfloppy                (1) - a simple floppy formatter for the GNOME
mbadblocks             (1) - tests a floppy disk, and marks the bad
                             blocks in the FAT
mformat                (1) - add an MSDOS filesystem to a low-level
                             formatted floppy disk
```

在输出中，每一行的第一个字段是手册页的名称，第二个字段显示部分（section）。注意，带有-k 选项的 man 命令与 apropos 命令在功能上基本是一致的。

5.3.5 whatis——显示命令的简要描述

whatis 程序显示匹配具体关键字的手册页的名字和一行描述。

```
[me@linuxbox ~]$ whatis ls
ls                  (1) - list directory contents
```

最难以忍受的参考手册

我们已经看到，Linux 和其他类 UNIX 系统中的手册文档是作为参考文档而非教程来使用的。很多手册文档都难以阅读，其中 bash 提供的手册页最为困难。在本书的编写过程中，我仔细复查，以确保覆盖了手册文档中的大部分主题。如果打印出来的话，将超过 80 页，而且排版非常紧凑，对一个初学者而言，这种结构是毫无意义的。

另一方面，bash 手册文档非常准确，也非常简要，同时也相当完备。所以，如果有胆量就去尝试一下，并期待有一天能读懂它。

5.3.6 info——显示程序的 info 条目

GNU 项目提供了 info 页面来代替手册文档。info 页面可通过 info 阅读器来显示。info 页面使用超链接，这与网页结构很相似。实例如下。

```
File: coreutils.info, Node: ls invocation, Next: dir invocation, Up:
Directory listing

10.1 'ls': List directory contents
==================================

The 'ls' program lists information about files (of any type, including
directories). Options and file arguments can be intermixed arbitrarily, as
usual.

   For non-option command-line arguments that are directories, by default 'ls'
lists the contents of directories, not recursively, and omitting files with
names beginning with '.'. For other non-option arguments, by default 'ls'
lists just the filename. If no non-option argument is specified, 'ls' operates
on the current directory, acting as if it had been invoked with a single
argument of '.'.

   By default, the output is sorted alphabetically, according to the
--zz-Info: (coreutils.info.gz)ls invocation, 63 lines --Top---------
```

info 程序读取 info 文件，该文件是树形结构，分为各个单独的节点，每一个节点包含一个主题。info 文件包含的超链接可以实现节点间的跳转。通过前置星号可以识别超链接，将光标放在超链接上并按 Enter 键，可以激活它。

可以通过输入 info 以及程序名（可选的）来调用 info。表 5-2 列出了显示 info 页面时，用于控制阅读器的命令。

表 5-2　info 命令

命令	功能
?	显示命令帮助
PAGE UP or BACKSPACE	返回上一页
PAGE DOWN or Spacebar	翻到下一页
n	Next——显示下一个节点
p	Previous——显示上一个节点
u	Up——显示目前显示节点的父节点（通常是一个菜单）
ENTER	进入光标所指的超链接
q	退出

到目前为止，我们讨论的大部分命令行程序都是 GNU 项目 coreutils 包的一部分，输入以下内容可以看到更多的信息。

```
[me@linuxbox ~]$ info coreutils
```

我们将会看到一个菜单页面，该菜单页面包含了 coreutils 包提供的每个程序的文档的超链接。

5.3.7 README 和其他程序文档文件

系统中安装的很多软件包都有自己的文档文件，它们存放在/usr/share/doc 目录中。其中大部分文档文件是以纯文本格式存储的，因此可以用 less 命令来查看。有些文件是 HTML 格式，并且可以用 Web 浏览器来查看。我们可能会遇到一些以.gz 扩展名结尾的文件。这表明它们是使用 gzip 压缩程序压缩过的。gzip 包包含一个特殊的 less 版本，称之为 zless。zless 可以显示由 gzip 压缩的文本文件的内容。

5.4 使用别名创建自己的命令

现在我们可以尝试编写程序了！我们可以使用 alias 命令来创建自己的命令。

但是在开始之前，我们需要展示一个命令行的小技巧。通过使用分号来分隔多条命令，就可以将多条命令输入在一行中。其工作方式如下：

command1;command2;command3...

我们将要使用的例子如下：

```
[me@linuxbox ~]$ cd /usr; ls; cd -
bin  games    kerberos lib64   local share tmp
etc  include  lib      libexec sbin  src
/home/me
[me@linuxbox ~]$
```

可以看到，我们将 3 条命令放置在同一行中。首先我们将当前目录改变成/usr，然后列出这个目录内容，最后返回到原始目录（使用 cd-）。那么程序结束的位置恰恰是开始的位置。现在，我们通过使用 alias 命令将以上命令整合成一条新的命令。首先要为新命令虚构出一个名称，试试名称 test。不过输入前，我们最好检查一下名称 test 是否已经被使用过了。为此，我们可以再次使用 type 命令。

```
[me@linuxbox ~]$ type test
test is a shell builtin
```

啊哦！这个名字已经用过了，试试 foo。

```
[me@linuxbox ~]$ type foo
bash: type: foo: not found
```

太好了！foo 还没有被使用。下面创建新命令的别名。

```
[me@linuxbox ~]$ alias foo='cd /usr; ls; cd -'
```

注意这个命令的结构。

alias *name='string'*

在 alias 命令之后输入 name，紧接着是一个等号（没有空格），等号之后是一个用单引号括起来的字符串，该字符串中的内容将赋值给 name。定义好的别名可以用在 shell 期待的任何地方。

尝试如下命令：

```
[me@linuxbox ~]$ foo
bin  games    kerberos  lib64   local share tmp
etc  include  lib       libexec sbin  src
/home/me
[me@linuxbox ~]$
```

也可以再次使用 type 命令来查看别名。

```
[me@linuxbox ~]$ type foo
foo is aliased to 'cd /usr; ls ; cd -'
```

要删除别名，可以使用 unalias 命令，如下所示。

```
[me@linuxbox ~]$ unalias foo
[me@linuxbox ~]$ type foo
bash: type: foo: not found
```

尽管我们有意避免使用已经存在的命名名称来给我们的别名命名，但有时也会期待这么做。这样做的目的是，为每一个经常调用的命令添加一个普遍会用到的选项。例如，前面讲到的为 ls 命令添加别名，以添加颜色支持。

```
[me@linuxbox ~]$ type ls
ls is aliased to 'ls --color=tty'
```

要查看在环境中定义的所有别名，可是使用不带参数的 alias 命令。以下是 Fedora 系统默认定义的一些别名。试着弄明白它们是干什么的。

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
```

在命令行定义别名还有一个小问题。当 shell 会话结束时，这些别名也随之消失了。在随后的章节中，我们将学习如何向文件中添加别名。每一次登录系统时，这些文件都会建立系统环境。现在，我们已经开始迈出了第一步，纵然它微不足道，可毋庸置疑的是，现在我们已经走进了 shell 编程的世界。

5.5 温故以求新

我们已经学习了如何查找命令文档，现在我们就来查看之前遇到的所有命令的文档，学习一下这些命令的其他选项，并练习使用。

第 6 章 重定向

本章我们将要探讨命令行最酷的功能——I/O 重定向。I/O 是输入/输出（input/output）的缩写。这个功能可以把命令行的输入重定向为从文件中获取内容，也可以把命令行的输出结果重定向到文件中。如果我们将多个命令行关联起来，将形成非常强大的命令——管道。接下来，我们将通过介绍以下命令来展示这一功能。

- cat：合并文件。
- sort：对文本行排序。
- uniq：报告或删除文件中重复的行。
- wc：打印文件中的换行符、字和字节的个数。
- grep：打印匹配行。
- head：输出文件的第一部分内容。
- tail：输出文件的最后一部分内容。

- tee：读取标准输入的数据，并将其内容输出到标准输出和文件中。

6.1 标准输入、标准输出和标准错误

到目前为止，我们使用过的很多程序生成了不同种类的输出。这些输出通常包含两种类型。一种是程序运行的结果，即该程序生成的数据；另一种是状态和错误信息，表示程序当前的运行情况。比如输入 ls 命令，屏幕上将显示它的运行结果以及它的相关错误信息。

与 UNIX“一切都是文件”的思想一致，类似 ls 的程序实际上把它们的运行结果发送到了一个称为标准输出（standard output，通常表示为 stdout）的特殊文件中，它们的状态信息则发送到了另一个称为标准错误（standard error，表示为 stderr）的文件中。默认情况下，标准输出和标准错误都将被链接到屏幕上，并且不会被保存在磁盘文件中。

另外，许多程序从一个称为标准输入（standard input，表示为 stdin）的设备来得到输入。默认情况下，标准输入连接到键盘。

I/O 重定向功能可以改变输出内容发送的目的地，也可以改变输入内容的来源地。通常来说，输出内容显示在屏幕上，输入内容来自于键盘。但是使用 I/O 重定向功能可以改变这一惯例。

6.1.1 标准输出重定向

I/O 重定向功能可以重新定义标准输出内容发送到哪里。使用重定向操作符“>”，后面接文件名，就可以把标准输出重定向到另一个文件中，而不是显示在屏幕上。为什么我们需要这样做呢？它主要用于把命令的输出内容保存到一个文件中。比如，我们可以按照下面的形式把 ls 命令的输出保存到 ls-output.txt 文件中，而不是输出到屏幕上。

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

这里，我们将创建/usr/bin 目录的一个长列表信息，并把这个结果输出到 *ls-output.txt* 文件中。检查下该命令被重定向的输出内容。

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1  me  me   167878 2012-02-01 15:07 ls-output.txt
```

这是一个不错的大型文本文件。如果使用 less 命令查看这个文件，我们可以看到 ls-output.txt 文件确实包含了 ls 命令的执行结果。

```
[me@linuxbox ~]$ less ls-output.txt
```

现在，让我们重复重定向测试，但是这次做一点变换。我们把目录名称换成一个不存在的目录。

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt
ls: cannot access /bin/usr: No such file or directory
```

我们会收到一条错误信息。因为我们指定的是一个不存在的目录/bin/usr，所以这个错误信息是正确的。但是为什么这个错误信息显示在屏幕上，而不是重定向到ls-output.txt文件中呢？原因是ls程序并不会把它运行的错误信息发送到标准输出文件中。而是与大多数写得很好的UNIX程序一样，它把错误信息发送到标准错误文件中。因为我们只重定向了标准输出，并没有重定向标准错误，所以这个错误信息仍然输出到屏幕上。稍后我们将讲述如何重定向标准错误，但是首先，先让我们看看这个输出文件发生了什么变化。

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me   me    0 2012-02-01 15:08 ls-output.txt
```

当前这个文件大小为零！这是因为当使用重定向符“>”来重定向标准输出时，目的文件通常会从文件开头部分重新改写。由于ls命令执行后没有输出任何内容，只是显示一条错误信息，所以重定向操作开始重新改写这个文件，并在出现错误的情况下停止操作，最终导致了该文件内容被删除。事实上，如果我们需要删除一个文件内容（或者创建一个新的空文件），可以采用这样的方式。

```
[me@linuxbox ~]$ > ls-output.txt
```

仅仅使用了重定向符，并在它之前不加任何命令，就可以删除一个已存在的文件内容或者创建一个新的空文件。

那么，我们如何能够不从文件的首位置开始覆盖文件，而是从文件的尾部开始添加输出内容呢？我们可以使用重定向符“>>”来实现，比如：

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
```

使用重定向符>>将使得输出内容添加在文件的尾部。如果这个文件并不存在，将与操作符>的作用一样创建这个文件。下面验证一下该操作符。

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me   me    503634 2012-02-01 15:45 ls-output.txt
```

重复执行这条命令三次，系统将最终生成一个为原来三倍大小的输出文件。

6.1.2 标准错误重定向

标准错误的重定向并不能简单地使用一个专用的重定向符来实现。要实现标准错误的重定向，不得不提到它的文件描述符（file descriptor)。一个程序可以把生成的输出内容发送到任意文件流中。如果把这些文件流中的前三个分别对应标准输入文件、标准输出文件和标准错误文件，那么 shell 将在内部用文件描述符分别索引它们为 0、1 和 2。shell 提供了使用文件描述符编号来重定向文件的表示法。由于标准错误等同于文件描述符 2，所以可以使用这种表示法来重定向标准错误。

```
[me@linuxbox ~]$ ls -l /bin/usr 2> ls-error.txt
```

文件描述符“2”紧放在重定向符之前，将标准错误重定向到 ls-error.txt 文件中。

6.1.3 将标准输出和标准错误重定向到同一个文件

在许多情况下，我们会希望把一个命令的所有输出内容都放在同一个独立的文件中。为此，我们必须同时重定向标准输出和标准错误。有两种方法可以满足要求。第一种是传统的方法，在旧版本的 shell 中使用。

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt 2>&1
```

使用这个方法，将执行两个重定向操作。首先重定向标准输出到 ls-output.txt 文件中，然后使用标记符 2>&1 把文件描述符 2（标准错误）重定向到文件描述符 1（标准输出）中。

注意 这些重定向操作的顺序是非常重要的。标准错误的重定向操作通常发生在标准输出重定向操作之后，否则它将不起作用。在上面的例子中，>ls-output.tx t 2>&1 把标准错误重定向到 *ls-output.txt* 文件中，但是如果顺序改变为 2>&1>*ls-output.txt*，那么标准错误将会重定向到屏幕上。

最近的 bash 版本提供了效率更高的第二种方法来实现这一联合的重定向操作。

```
[me@linuxbox ~]$ ls -l /bin/usr &> ls-output.txt
```

在这个例子中，只使用一个标记符“&>”就把标准输出和标准错误都重定

向到了 *ls-output.txt* 文件中。

6.1.4 处理不想要的输出

有时候“沉默是金”，命令执行后我们并不希望得到输出，而是想把这个输出丢弃，尤其是在输出错误和状态信息的情况下更为需要。系统提供了一种方法，即通过把输出重定向到一个称为/dev/null 的特殊文件中来实现它。这个文件是一个称为位桶（bit bucket）的系统设备，它接受输入但是不对输入进行任何处理。以下命令可以用来抑制（即隐藏）一个命令的错误信息。

```
[me@linuxbox ~]$ ls -l /bin/usr 2> /dev/null
```

UNIX 文化中的/DEV/NULL

位桶（bit bucket）是一个古老的 UNIX 概念，由于它的普适性，它出现在 UNIX 文化的很多地方。因此当某人说他正把你的意见发送到“dev null”的时候，现在你知道他是什么意思了。你可以在 http://en.wikipedia.org/wiki/Dev/null 中查看维基百科的相关文章，了解更多的相关示例。

6.1.5 标准输入重定向

到目前为止，我们还没有接触过使用标准输入的命令（实际上已经遇到了，稍后将揭晓这个谜底），接下来我们先介绍一个命令。

cat——合并文件

cat 命令读取一个或多个文件，并把它们复制到标准输出文件中，格式如下。

```
cat [file...]
```

在大多数情况下，你可以认为 cat 命令和 DOS 中的 TYPE 命令类似。使用它显示文件而不需要分页，例如：

```
[me@linuxbox ~]$ cat ls-output.txt
```

将显示 ls-output.txt 文件的内容。cat 经常用来显示短的文本文件。由于 cat 可以接受多个文件作为输入参数，所以它也可以用来把文件连接在一起。假设我们下载了一个很大的文件，它已被拆分为多个部分（Usenet 上的多媒体文件经常采用拆分这种方式），现在我们想要把各部分连接在一起，并还原为原来的文件。如果这些文件命名为

movie.mpeg.001 movie.mpeg.002...movie.mpeg.099

我们可以使用这个命令让它们重新连接在一起。

```
[me@linuxbox ~]$ cat movie.mpeg.0* > movie.mpeg
```

通配符一般都是按照顺序来扩展的，因此这些参数将按正确的顺序来排列。

虽然这样很好，但是这跟标准输入有什么关系呢？确实没有任何关系，但是我们可以试试其他的情况。如果输入 cat 命令却不带任何参数，会出现什么样的结果呢？

```
[me@linuxbox ~]$ cat
```

没有任何结果——它只是停在那边不动，好像它已经挂起了。看起来好像是这样的，但是它实际上正在执行我们期望它做的事情。

如果 cat 命令没有给定任何参数，它将从标准输入读取内容。由于标准输入在默认情况下是连接到键盘，所以实际上它正在等待着从键盘输入内容！

试下这个。

```
[me@linuxbox ~]$ cat
The quick brown fox jumped over the lazy dog.
```

下一步，按下 Ctrl-D（按住 Ctrl 键同时按下 D），告知 cat 命令它已经达到了标准输入的文件尾（end-of-file，EOF）。

```
[me@linuxbox ~]$ cat
The quick brown fox jumped over the lazy dog.
The quick brown fox jumped over the lazy dog.
```

在缺少文件名参数的情况下，cat 将把标准输入内容复制到标准输出文件中，因此我们将看到文本行重复显示。用这种方法我们可以创建短的文本文件。如果想要创建一个名叫 lazy_dog.txt 的文件，文件中包含之前例子中的文本内容，我们可以这样做：

```
[me@linuxbox ~]$ cat > lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

在 cat 命令后输入想要放在文件中的文本内容。记住在文件结束时按下 Ctrl-D。使用这个命令行，相当于执行了世界上最愚蠢的文字处理器！为了看到结果，我们可以使用 cat 命令再次把文件复制到标准输出文件中。

```
[me@linuxbox ~]$ cat lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

现在我们已经知道 cat 命令除了接受文件名参数之外，是如何接受标准输入的。接下来尝试一下标准输入的重定向。

```
[me@linuxbox ~]$ cat < lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

使用重定向符“<”，我们将把标准输入的源从键盘变为 lazy_dog.txt 文件。可以看到，得到的结果和只传递单个文件名参数的结果一样。和传输一个文件名参数的方式作对比，这种方式并不是特别的有用，但是可以用来说明把一个文件作为标准输入的源文件。还有其他的命令更好地使用了标准输入，稍后会讲到。

在继续学习下面内容之前，我们可以查看 cat 命令的手册文档，因为它有几个有趣的选项。

6.2 管道

命令从标准输入到读取数据，并将数据发送到标准输出的能力，是使用了名为管道的 shell 特性。使用管道操作符“|”（竖线）可以把一个命令的标准输出传送到另一个命令的标准输入中。

```
Command1 | command2
```

为了充分证明这一点，我们需要一些命令。还记得之前说过有一条已知的命令可以接受标准输入吗？它就是 less 命令。使用 less 命令可以分页显示任意命令的输入，该命令将它的结果发送到标准输出。

```
[me@linuxbox ~]$ ls -l /usr/bin | less
```

这相当方便！通过使用该技术，可以很方便地检查任意一条生成标准输出的命令的运行结果。

6.2.1 过滤器

管道功能经常用来对数据执行复杂的操作。也可以把多条命令合在一起构成一个管道。这种方式中用到的命令通常被称为过滤器（filter)。过滤器接受输入，按照某种方式对输入进行改变，然后再输出它。第一个要用到的命令是 sort。假设要把/bin 和/usr/bin 目录下的所有可执行程序合并成一个列表，并且按照顺序排列，最后再查看这个列表。

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | less
```

由于我们指定了两个目录（/bin 和/usr/bin），ls 的输出将包含两个排好序的列表，每个对应一个目录。通过在管道中包含 sort 命令，我们改变输出数据，从而产生一个排好序的列表。

6.2.2 uniq——报告或忽略文件中重复的行

uniq 命令经常和 sort 命令结合使用。uniq 可以接受来自于标准输入或者一个单一文件名参数对应的已排好序的数据列表（可以查看 uniq 命令的 man 页面获取详细信息）。默认情况下，该命令删除列表中的所有重复行。因此，在管道中添加 uniq 命令，可以确保所有的列表都没有重复行（即在/bin 和/usr/bin 目录下都出现的相同名字的任意程序）。

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | less
```

在这个例子中，我们使用了 uniq 命令来删除来自 sort 命令输出内容中的任意重复行。如果反过来想要查看重复行的列表，可以在 uniq 命令后面添加-d 选项，如下所示。

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq -d | less
```

6.2.3 wc——打印行数、字数和字节数

wc（字数统计，word count）命令用来显示文件中包含的行数、字数和字节数。例如：

```
[me@linuxbox ~]$ wc ls-output.txt
 7902   64566 503634 ls-output.txt
```

在这个例子中，我们打印输出了三个数据，即 ls-output.txt 文件中包含的行数、字数和字节数。和前面的命令一样，如果在执行 wc 时没有输入输入命令行参数，它将接受标准输入内容。-l 选项限制命令只报告行数，把它添加在管道中可以很方便地实现计数功能。如果我们要查看已排好序的列表中的条目数，可以按以下方式输入。

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | wc -l
2728
```

6.2.4 grep——打印匹配行

grep 是一个功能强大的程序，它用来在文件中查找匹配文本，其使用方式如下。

```
grep pattern [file...]
```

当 grep 在文件中遇到“模式”的时候，将打印出包含该模式的行。grep 能够匹配的模式内容可以是非常复杂的，不过这里，我们只关注简单文本的匹配。在第 19 章，我们将介绍“正则表达式（regular expression）”的高级模式。

如果想我们从列出的程序中搜索出文件名中包含 zip 的所有文件，该搜索将获悉系统中与文件压缩相关的程序，操作如下。

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

grep 存在一对方便的选项：-i，该选项使得 grep 在搜索时忽略大小写（通常情况下，搜索是区分大小写的）；-v，该选项使得 grep 只输出和模式不匹配的行。

6.2.5 head/tail——打印文件的开头部分/结尾部分

有的时候，你并不需要命令输出的所有内容，可能只是需要开头几行或者最后几行。head 命令将输出文件的前 10 行，tail 命令则输出文件的最后 10 行。默认情况下，这两条命令都是输出文件的 10 行内容，不过可以使用-n 选项来调整输出的行数。

```
[me@linuxbox ~]$ head -n 5 ls-output.txt
total 343496
-rwxr-xr-x 1 root root      31316 2011-12-05 08:58 [
-rwxr-xr-x 1 root root       8240 2011-12-09 13:39 411toppm
-rwxr-xr-x 1 root root     111276 2011-11-26 14:27 a2p
-rwxr-xr-x 1 root root      25368 2010-10-06 20:16 a52dec
[me@linuxbox ~]$ tail -n 5 ls-output.txt
-rwxr-xr-x 1 root root       5234 2011-06-27 10:56 znew
-rwxr-xr-x 1 root root        691 2009-09-10 04:21 zonetab2pot.py
-rw-r--r-- 1 root root        930 2011-11-01 12:23 zonetab2pot.pyc
-rw-r--r-- 1 root root        930 2011-11-01 12:23 zonetab2pot.pyo
lrwxrwxrwx 1 root root          6 2012-01-31 05:22 zsoelim -> soelim
```

这些命令选项也可以应用在管道中。

```
[me@linuxbox ~]$ ls /usr/bin | tail -n 5
znew
```

```
zonetab2pot.py
zonetab2pot.pyc
zonetab2pot.pyo
zsoelim
```

tail 中有一个选项用来实时查看文件，该选项在观察正在被写入的日志文件的进展状态时很有用。在下面的例子中，我们将观察/var/log 目录下的 messages 文件。因为/var/log/messages 文件可能包含安全信息，所以在一些 Linux 发行版本中，需要超级用户的权限才能执行该操作。

```
[me@linuxbox ~]$ tail -f /var/log/messages
Feb  8 13:40:05 twin4 dhclient: DHCPACK from 192.168.1.1
Feb  8 13:40:05 twin4 dhclient: bound to 192.168.1.4 -- renewal in 1652
seconds.
Feb  8 13:55:32 twin4 mountd[3953]: /var/NFSv4/musicbox exported to both
192.168.1.0/24 and twin7.localdomain in 192.168.1.0/24,twin7.localdomain
Feb  8 14:07:37 twin4 dhclient: DHCPREQUEST on eth0 to 192.168.1.1 port 67
Feb  8 14:07:37 twin4 dhclient: DHCPACK from 192.168.1.1
Feb  8 14:07:37 twin4 dhclient: bound to 192.168.1.4 -- renewal in 1771
seconds.
Feb  8 14:09:56 twin4 smartd[3468]: Device: /dev/hda, SMART Prefailure
Attribute: 8 Seek_Time_Performance changed from 237 to 236
Feb  8 14:10:37 twin4 mountd[3953]: /var/NFSv4/musicbox exported to both
192.168.1.0/24 and twin7.localdomain in 192.168.1.0/24,twin7.localdomain
Feb  8 14:25:07 twin4 sshd(pam_unix)[29234]: session opened for user me by
(uid=0)
Feb  8 14:25:36 twin4 su(pam_unix)[29279]: session opened for user root by
me(uid=500)
```

使用-f 选项，tail 将持续监视这个文件，一旦添加了新行，新行将会立即显示在屏幕上。该动作在按下 Ctrl-C 后停止。

6.2.6 tee——从 stdin 读取数据，并同时输出到 stdout 和文件

为了和我们的管道隐喻保持一致，Linux 提供了一个叫做 tee 的命令，就好像安装了一个“T”在管道上。tee 程序读取标准输入，再把读到的内容复制到标准输出（允许数据可以继续向下传递到管道中）和一个或更多的文件中去。当在某个中间处理阶段来捕获一个管道中的内容时，会很有用。这里我们重复使用之前的一个例子，这次在使用 grep 命令过滤管道内容之前，我们先使用 tee 命令来获取整个目录列表并输出到 ls.txt 文件中，具体操作如下。

```
[me@linuxbox ~]$ ls /usr/bin | tee ls.txt | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
```

```
zipgrep
zipinfo
zipnote
zipsplit
```

6.3 本章结尾语

和以前一样，请查看本章介绍的各个命令的相关文档。本章只介绍了这些命令最基本的用法，它们都还有很多其他有趣的选项。在有一定 Linux 使用经验的时候，我们将会发现命令行的重定向功能对于解决某些特定的问题相当有用。很多命令使用了标准输入和输出，而且几乎所有的命令行程序都使用了标准错误来显示它们的提示性信息。

富有想象力的 Linux

每当被问到 Windows 和 Linux 的区别时，我经常通过用玩具打比方的方式来进行解释。

Windows 就像是 Game Boy 游戏机。你去商店买了一个全新的游戏机。你把它带回家，启动它，开始玩这个游戏机。漂亮的画面，可爱的声音。但是不久，你对这款游戏机玩腻了，于是你回到商店，买了另一款游戏机。这个过程一遍一遍地重复着。最后，你再次回到商店，对柜台后的售货员说“我想要一款可以玩这个游戏的游戏机！”但是却被告知因为没有针对它的“市场需求”，所以并不存在这种游戏机。然后你会说“但是我只需要更换这一个东西就行了”。柜台后的售货员将会对你说，你不能更换它。这个游戏机盒子都是完全密封好的。你发现你的玩具选择范围被限定了，你只能选择由别人决定的认为你需要的游戏，并没有其他更多的游戏可选。

而另一方面，Linux 就像是世界上最大的建筑拼装玩具。你打开它，发现它只是一个超大的零件集——一大堆的钢架、螺丝钉、螺帽、齿轮、滑轮组以及马达，另附上可拼装的一些参考样式图案。你开始玩这个玩具。你拼装了一种参考样式后，再接着拼装另一种。不久，你发现你可以拼装出自己想要的样式。你不再需要非得回到商店，因为你已经有你需要的所有东西。这个建筑拼装玩具可以呈现出你想象的形状，它可以实现你所想要的。

当然，选择哪个玩具是你自己的事情，你觉得你会更加钟情于哪种玩具呢？

第 7 章

透过 shell 看世界

在本章，我们将介绍在按下 Enter 键时，命令行中发生的一些“神奇”事情。虽然我们会介绍 shell 的几个有趣而复杂的特性，但是我们只使用一条新命令来处理。

- echo：显示一行文本。

7.1 扩展

每次输入命令行按下 Enter 键时，bash 都会在执行命令之前对文本进行多重处理。前面已经见过一个简单的字符序列（比如*）在 shell 中被识别为多种意思的几个例子。产生这个结果的处理过程称为扩展（expansion）。有了扩展功能，在输入内容后，这些内容将在 shell 对其执行之前被扩展成其他内容。为了证明这点，让我们先来看看 echo 命令。echo 是 shell 的一个内置命令，它执行的任务非常简单，即把文本参数内容打印到标准输出。

```
[me@linuxbox ~]$ echo this is a test
this is a test
```

这个例子相当简单，传递给 echo 的任何参数都将显示出来。让我们再看另一个例子。

```
[me@linuxbox ~]$ echo *
Desktop Documents ls-output.txt Music Pictures Public Templates Videos
```

刚刚发生了什么？为什么 echo 不是输出“*”呢？回想一下之前我们对通配符的使用。“*”字符意味着“匹配文件名中的任意字符”，但是之前我们并没有讨论 shell 是如何实现这个功能的。答案很简单，shell 会在执行 echo 命令前把“*”字符扩展成其他内容（在这个例子中，扩展为当前工作目录下的所有文件名）。在按下 Enter 键的时候，shell 会在执行命令前自动扩展命令行中所有符合条件的字符，因此 echo 命令将不可能看到“*”字符，只能看到“*”字符扩展后的结果。知道了这些，我们就会发现 echo 命令输出的正是预期的结果。

7.1.1 路径名扩展

通过使用通配符来实现扩展的机制称为路径名扩展（pathname expansion）。试试在前面章节中使用过的一些技术，将会发现它们实际上就是扩展。下面给定一个主目录，如下所示：

```
[me@linuxbox ~]$ ls
Desktop   ls-output.txt  Pictures   Templates
Documents Music          Public     Videos
```

执行下面的扩展：

```
[me@linuxbox ~]$ echo D*
Desktop Documents
```

以及

```
[me@linuxbox ~]$ echo *s
Documents Pictures Templates Videos
```

甚至是

```
[me@linuxbox ~]$ echo [[:upper:]]*
Desktop Documents Music Pictures Public Templates Videos
```

查看除主目录之外的目录：

```
[me@linuxbox ~]$ echo /usr/*/share
/usr/kerberos/share /usr/local/share
```

隐藏文件的路径名扩展

众所周知，文件名以一个“.”点字符开头的文件都将被隐藏。路径名扩展功能也遵守这个规则。类似 echo *这样的扩展并不能显示隐藏的文件。

乍一看，好像可以通过在扩展的模式中以一个点字符开头来包含隐藏文件，如下所示。

```
echo .*
```

这似乎是可行的。但是如果我们仔细地检查结果，会发现文件名“.”和“..”也将出现在结果中。由于这两个名字分别指的是当前的工作目录以及当前目录的父目录，使用这种模式匹配可能会生成不正确的结果。执行命令行 ls –d .*|less 可以发现这个结果是不正确的。

在这种情况下，要正确地执行路径名扩展，必须采用一种更精确些的模式，它会正确地工作。

```
ls -d .[!.]?*
```

这种模式将扩展为以一个点字符开头的所有文件名，文件名中并不包含第二个点字符，但包含至少一个额外的字符，后面也可能还跟着其他的字符。

7.1.2　波浪线扩展

回顾前面对 cd 命令的介绍，你会发现波浪线字符（～）具有特殊的含义。如果把它用在一个单词的开头，那么它将被扩展为指定用户的主目录名；如果没有指定用户命名，则扩展为当前用户的主目录。

```
[me@linuxbox ~]$ echo ~
/home/me
```

如果有用户 foo 这个账户，那么：

```
[me@linuxbox ~]$ echo ~foo
/home/foo
```

7.1.3　算术扩展

shell 支持通过扩展来运行算术表达式。这允许我们把 shell 提示符当作计算器来使用。

```
[me@linuxbox ~]$ echo $((2 + 2))
4
```

算术扩展使用如下格式。

```
$((expression))
```

其中，expression 是指包含数值和算术操作符的算术表达式。

算术扩展只支持整数（全是数字，没有小数），但是可以执行很多不同的运算。表 7-1 列出了一些支持的操作符。

表 7-1 算术运算符

运算符	描述
+	加
-	减
*	乘
/	除（但是记住，因为扩展只支持整数运算，所以结果也是整数）
%	取余，即余数
**	取幂

空格在算术表达式中是没有意义的，而且表达式是可以嵌套的。例如把 5^2 和 3 相乘。

```
[me@linuxbox ~]$ echo $(($((5**2)) * 3))
75
```

你可以使用一对括号来组合多个子表达式。通过该技术，可以把上面的例子重写，用一个扩展来代替两个，可以得到同样的结果：

```
[me@linuxbox ~]$ echo $(((5**2) * 3))
75
```

下面的例子使用了除运算符和取余运算符，注意整数相除的结果。

```
[me@linuxbox ~]$ echo Five divided by two equals $((5/2))
Five divided by two equals 2
[me@linuxbox ~]$ echo with $((5%2)) left over.
with 1 left over.
```

在第 34 章我们将更详细地介绍算术扩展。

7.1.4 花括号扩展

花括号扩展（brace expansion）可能算是最奇怪的扩展方式了。有了它，你可以按照花括号里面的模式创建多种文本字符串。实例如下。

```
[me@linuxbox ~]$ echo Front-{A,B,C}-Back
Front-A-Back Front-B-Back Front-C-Back
```

用于花括号扩展的模式信息可以包含一个称为前导字符（preamble）的开头部分和一个称为附言（postscript）的结尾部分。花括号表达式本身可以包含一系列逗号分隔的字符串，也可以包含一系列整数或者单个字符。这里的模式信息不能包含内嵌的空白。下面的例子使用了一系列的整数。

```
[me@linuxbox ~]$ echo Number_{1..5}
Number_1 Number_2 Number_3 Number_4 Number_5
```

下面输出一系列逆序排列的字母。

```
[me@linuxbox ~]$ echo {Z..A}
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
```

花括号扩展支持嵌套。

```
[me@linuxbox ~]$ echo a{A{1,2},B{3,4}}b
aA1b aA2b aB3b aB4b
```

那么花括号扩展一般应用在哪些地方呢？最普遍的应用是创建一系列的文件或者目录。比如说，摄影师有一个很大的图片集，想要按年份和月份来对这些图片进行分组，那么要做的第一件事就是创建一系列以年月格式命名的目录。这样，这些目录名将会按照年代顺序排列，输出目录的一个完整的列表。但是这样做工作量大，而且容易出错。为此我们可以这样操作。

```
[me@linuxbox ~]$ mkdir Pics
[me@linuxbox ~]$ cd Pics
[me@linuxbox Pics]$ mkdir {2009..2011}-0{1..9} {2009..2011}-{10..12}
[me@linuxbox Pics]$ ls
2009-01   2009-07   2010-01   2010-07   2011-01   2011-07
2009-02   2009-08   2010-02   2010-08   2011-02   2011-08
2009-03   2009-09   2010-03   2010-09   2011-03   2011-09
2009-04   2009-10   2010-04   2010-10   2011-04   2011-10
2009-05   2009-11   2010-05   2010-11   2011-05   2011-11
2009-06   2009-12   2010-06   2010-12   2011-06   2011-12
```

相当巧妙!

7.1.5　参数扩展

本章我们只是简要地介绍参数扩展（parameter expansion），在之后的章节中我们将会更深入地介绍它。参数扩展用在 shell 脚本中比直接用在命令行中更为有用。它的许多特性与系统存储小块数据以及给每个小块数据命名的性能有关。很多这样的小块数据（称为变量[variable]会更合适）可用于扩展。例如，命名为 USER 的变量包含你的用户名，为了触发参数扩展，并显示出 USER 的内容，你可以进行如下操作。

```
[me@linuxbox ~]$ echo $USER
me
```

想要查看可用的变量列表，试试如下操作。

```
[me@linuxbox ~]$ printenv | less
```

你可能已经注意到，对于其他的扩展类型来说，如果你误输入了一个模式，就不会发生扩展，这时 echo 命令将只是显示这些误输入的模式信息。但是对于参数扩展来说，如果变量名拼写错误，仍然会进行扩展，只不过结果是输出一个空字符串而已，如下所示。

```
[me@linuxbox ~]$ echo $SUER

[me@linuxbox ~]$
```

7.1.6 命令替换

命令替换可以把一个命令的输出作为一个扩展模式使用，如下所示。

```
[me@linuxbox ~]$ echo $(ls)
Desktop Documents ls-output.txt Music Pictures Public Templates Videos
```

我最喜欢的一种用法如下。

```
[me@linuxbox ~]$ ls -l $(which cp)
-rwxr-xr-x 1 root root 71516 2012-12-05 08:58 /bin/cp
```

这里，把 which cp 命令的运行结果作为 ls 命令的一个参数，因此我们无需知道 cp 程序所在的完整路径就能获得 cp 程序对应的列表。这个功能并不只是局限于简单的命令，也可以应用于整个管道中（只不过只显示部分输出内容）。

```
[me@linuxbox ~]$ file $(ls /usr/bin/* | grep zip)
/usr/bin/bunzip2:     symbolic link to 'bzip2'
/usr/bin/bzip2:       ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV
), dynamically linked (uses shared libs), for GNU/Linux 2.6.9, stripped
/usr/bin/bzip2recover: ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.9, stripped
/usr/bin/funzip:      ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV
), dynamically linked (uses shared libs), for GNU/Linux 2.6.9, stripped
/usr/bin/gpg-zip:     Bourne shell script text executable
/usr/bin/gunzip:      symbolic link to '../../bin/gunzip'
/usr/bin/gzip:        symbolic link to '../../bin/gzip'
/usr/bin/mzip:        symbolic link to 'mtools'
```

在这个例子中，管道的输出为 file 命令的参数列表。

在早期的 shell 程序中，存在命令替换的另一种语法格式，bash 也支持这种格式。它用反引号代替美元符号和括号，具体如下所示。

```
[me@linuxbox ~]$ ls -l `which cp`
-rwxr-xr-x 1 root root 71516 2012-12-05 08:58 /bin/cp
```

7.2 引用

我们已经知道，shell 有多种方式可以执行扩展，现在我们来学习如何控制扩展。先看下面的例子。

```
[me@linuxbox ~]$ echo this is a    test
this is a test
```

再看这个例子。

```
[me@linuxbox ~]$ echo The total is $100.00
The total is 00.00
```

在第一个例子中，shell 会对 echo 命令的参数列表进行单词分割（*word splitting*），去除多余的空白。在第二个例子中，因为$1 是一个未定义的变量，所以参数扩展将把$1 的值替换为空字符串。shell 提供了一种称为引用（quoting）的机制，用来有选择性地避免不想要的扩展。

7.2.1 双引号

我们要看的第一种引用类型是双引号（double quote）。如果把文本放在双引号中，那么 shell 使用的所有特殊字符都将失去它们的特殊含义，而被看成普通字符。字符“$”（美元符号）、“\”（反斜杠）、“`”（反引号）除外。这就意味着单词分割、路径名扩展、波浪线扩展和花括号扩展都将失效，但是参数扩展、算术扩展和命令替换仍然生效。使用双引号能够处理文件名中包含空白的情况。假设不幸地有一个名为 two words.txt 的文件，如果在命令行中使用该文件名，那么单词分割功能将把它当成两个独立的参数，而不是当成我们希望的单个参数，具体运行结果如下所示。

```
[me@linuxbox ~]$ ls -l two words.txt
ls: cannot access two: No such file or directory
ls: cannot access words.txt: No such file or directory
```

使用双引号可以阻止单词分割，得到预期的结果。另外，使用双引号甚至可以修复破损的文件名，参见下面的例子。

```
[me@linuxbox ~]$ ls -l "two words.txt"
-rw-rw-r-- 1 me me 18 2012-02-20 13:03 two words.txt
[me@linuxbox ~]$ mv "two words.txt" two_words.txt
```

看！现在我们就不需要一直输入那些让人讨厌的双引号了。

请记住，参数扩展、算术扩展和命令替换在双引号中依然生效：

```
[me@linuxbox ~]$ echo "$USER $((2+2)) $(cal)"
me 4 February 2012
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29
```

接下来，让我们看看双引号对字符替换的影响。我们首先深入了解一下单词分割是怎么工作的。在前面的例子中，我们已经看到单词分割去除文本中多余空白的情况，如下所示。

```
[me@linuxbox ~]$ echo this is a test
this is a test
```

默认情况下，单词分割会先查找是否存在空格、制表符以及换行（换行字符），然后把它们当作单词见的界定符（delimiter）。这就意味着没有用引号包含起来的空格、制表符和换行字符都不会被当成文本的一部分，而只是被当成分割符。因为它们把这些单词分割成不同的参数，所以例子中的命令行被识别为命令后面跟着 4 个不同的参数。但是如果加上双引号，单词分割功能将失效，嵌入的空格将不再被当成界定符，而是被当成参数的一部分，如下所示。

```
[me@linuxbox ~]$ echo "this is a     test"
this is a     test
```

一旦加上双引号，那么命令行将被识别为命令后面只跟着一个参数。

单词分割机制会把换行字符当成界定符，这一点在命令替换时将会产生微妙有趣的效果。参考下面的例子。

```
[me@linuxbox ~]$ echo $(cal)
February 2012 Su Mo Tu We Th Fr Sa 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29
[me@linuxbox ~]$ echo "$(cal)"
   February 2012
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29
```

在第一个例子中，没有加上引号的命令替换将导致命令行被识别为命令后面跟着 38 个参数；而在第二个例子中加了双引号，使得命令行被识别为命令后面只跟一个参数，这个参数包含着嵌入空格和换行字符。

7.2.2 单引号

如果我们希望抑制所有的扩展，那么应使用单引号。下面是不使用引号、使用双引号和使用单引号的情况对比。

```
[me@linuxbox ~]$ echo text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
text /home/me/ls-output.txt a b foo 4 me
[me@linuxbox ~]$ echo "text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER"
text ~/*.txt {a,b} foo 4 me
[me@linuxbox ~]$ echo 'text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER'
text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
```

可以看到，随着引用级别的加强，越来越多的扩展将被抑制。

7.2.3 转义字符

有时候我们只是想要引用单个字符。这种情况可以通过在该字符前加上反斜杠来实现。这里的反斜杠称为转义字符。转义字符经常在双引号中用来有选择性地阻止扩展。如下所示。

```
[me@linuxbox ~]$ echo "The balance for user $USER is: \$5.00"
The balance for user me is: $5.00
```

转义字符也常用来消除文件名中某个字符的特殊含义。比如，文件名中可以使用在 shell 中通常具有特殊含义的字符。这些字符包括“$”、“!”、“&”、空格等。要想在文件名中包含特殊字符，可执行如下操作。

```
[me@linuxbox ~]$ mv bad\&filename good_filename
```

如果想要显示反斜杠字符，可以通过使用两个反斜杠“\\”来实现。需要注意的是，单引号中的反斜杠将失去它的特殊含义，而只是被当成一个普通字符。

反斜杠转义字符序列（BLACKSLASH ESCAPE SEQUENCES）

反斜杠除了作为转义字符外，也是一种表示法的一部分，这种表示法代表称为控制码的某些特殊字符。ASCII 码表的前 32 个字符用来向电传打字类设备传送命令。其中有一些控制码很常见（比如制表符、退格符、换行符和回车符），但是其他的都不太常见（空字符、结束符和确认符），如表 7-2 所示。

表 7-2 反斜杠转义字符序列

转义字符	含义
\a	响铃（警告声—计算机发出哔哔声）
\b	退格
\n	新的一行（在类 UNIX 系统中，产生的是换行效果）
\r	回车
\t	制表

表中列出了一些常用的反斜杠转义字符序列。使用反斜杠来表示转义字符表示的理解来源于 C 语言，其他语言也采用了这种表示方法，包括 shell。

在 echo 命令中带上-e 选项，就能够解释转义字符序列。也可以将其放在“$‘’”中。在下面的例子中，只需要使用 sleep 命令（它是一个简单的程序，在等待指定的秒数之后就会退出），就可以创建一个简单的倒计时的计时器：

```
sleep 10; echo -e "Time's up\a"
```

也可以这样做：

```
sleep 10; echo "Time's up" $'\a'
```

7.3 本章结尾语

随着我们输入学习 shell，就会发现扩展和引用的使用频率逐渐多起来，所以我们有必要很好地理解它们的工作方式。事实上，甚至可以说它们是 shell 中最重要的主题。如果不能正确地理解扩展，那么 shell 将会一直是个神秘和让人困惑的资源，它的潜在能力也就被浪费了。

第 8 章

高级键盘技巧

我经常将 UNIX 戏称为“它是为喜欢敲键盘的人设计的操作系统”。当然，UNIX 中存在命令行的这一事实充分证明了这点。但是用户使用命令行时往往不喜欢敲入太多字，所以命令中存在很多类似 cp、ls、mv 和 rm 的短命令。

事实上，省事（laziness）（即用最少的击键次数执行最多的任务）是命令行最希望达到的目标之一。命令行的另一个目标是，用户在执行任务时手指无需离开键盘，用不使用鼠标。本章我们将学习可以令键盘使用得更快和更高效的 bash 功能。

我们将使用到以下命令。

- clear：清屏。
- history：显示历史列表的记录。

8.1 编辑命令行

bash 使用了一个名为 Readline 的库（供不同的应用程序共享使用的线程集合）

来实现命令行的编辑。在前面我们曾提到过相关内容。比如，通过箭头键移动光标。除此之外，bash 还有很多其他的功能，它们可以当作在工作中使用的附加工具。虽然并不要求你们学会所有的这些功能，但是学会其中的一些功能还是非常有用的。请选择自己需要的功能。

注意　下面的有些组合键（尤其对于那些使用了 Alt 键的组合键）可能会被 GUI（图形用户界面）识别为其他功能。当使用虚拟控制台时，所有的组合键应该能够正常工作。

8.1.1 光标移动

表 8-1 中列出了用来移动光标的组合键。

表 8-1　光标移动命令

组合键	作用
Ctrl-A	移动光标到行首
Ctrl-E	移动光标到行尾
Ctrl-F	光标向前移动一个字符；和右箭头键作用一样
Ctrl-B	光标向后移动一个字符，和左箭头键作用一样
Alt-F	光标向前移动一个字
Alt-B	光标向后移动一个字
Ctrl-L	清屏并把光标移到左上角；clear 命令可以完成相同的工作

8.1.2 修改文本

表 8-2 列出了用来编辑命令行字符的键盘指令。

表 8-2　文本编辑命令

组合键	作用
Ctrl-D	删除光标处的字符
Ctrl-T	使光标处的字符和它前面的字符对调位置
Alt-T	使光标处的字和它前面的字对调位置
Alt-L	把从光标到字尾的字符转换成小写字母形式
Alt-U	把从光标到字尾的字符转换成大写字母形式

8.1.3 剪切和粘贴（Killing and Yanking）文本

Readline 文档中使用术语 killing 和 yanking 来指代通常所说的剪切和粘贴。表 8-3 列出了用来剪切和粘贴的命令。被剪切的内容存放在一个称为 kill-ring 的缓冲区中。

表 8-3 剪切和粘贴命令

组合键	作用
Ctrl-K	剪切从光标到行尾的文本
Ctrl-U	剪切从光标到行首的文本
Alt-D	剪切从光标到当前词尾的文本
Alt-Backspace	剪切从光标到词头的文本。如果光标在一个单词的开头，则剪切前一个单词
Ctrl-Y	把 kill-ring 缓冲区中的文本粘贴到光标位置

元键

在 bash 帮助文档的“READLINE”部分可以查看 Readline 文档，在这里你将会看到元键（meta key）这个术语。它对应于现代键盘中的 Alt 键，不过也并不总是这样。

回到混沌时代（PC 时代前，UNIX 时代后），并不是每个人都有自己的计算机。当时的用户可能只有一台称为终端的设备。终端是一种通信设备，它包含一个文本显示屏、一个键盘以及一些用来显示文本字符和移动光标的电子器件。终端（通常通过串行电缆）连接到一台大型计算机或者大型计算机通信网。它有很多不同的品牌，因此有不同的键盘和不同的显示特性集。由于它们至少都能识别 ASCII 码，因此软件开发者想要编写符合最低标准的可移植的应用程序。UNIX 系统有一套非常巧妙的方法来处理这些终端以及它们不同的显示特性。因为 Readline 的开发者们不能确定是否存在一个专门的附加控制键，所以他们发明了一个，并把它称之为“元”。现代键盘上的 Alt 键相当于元键。如果你仍然在使用终端，则按下和释放 Esc 键和长按住 Alt 键的效果是相同的（对于 Linux 系统也是如此）。

8.2 自动补齐功能

shell 的一种称为“自动补齐”的机制为用户提供了很大的帮助。在输入命

令时，按 Tab 键将触发自动补齐功能。下面让我们看看它是如何工作的。假设用户目录如下。

```
[me@linuxbox ~]$ ls
Desktop   ls-output.txt  Pictures  Templates  Videos
Documents Music          Public
```

输入如下命令，但是不要按 Enter 键。

```
[me@linuxbox ~]$ ls l
```

此时按 Tab 键：

```
[me@linuxbox ~]$ ls ls-output.txt
```

观察 shell 是如何补齐这一行的。再看另一个例子，同样，也不要按 Enter 键。

```
[me@linuxbox ~]$ ls D
```

按下 Tab 键：

```
[me@linuxbox ~]$ ls D
```

没有自动补齐——只有哔哔声。这是因为字母 D 和目录中一个以上的名称匹配。要让自动补齐功能生效，要保证输入的内容不模棱两可，即必须是确定性的。如果我们继续输入：

```
[me@linuxbox ~]$ ls Do
```

此时按下 Tab：

```
[me@linuxbox ~]$ ls Documents
```

自动补齐功能这次生效了。

这个例子给出的是路径名的自动补齐，这也是最常用的方式。自动补齐也可以针对变量（如果单词以$开头）、用户名（如果单词以~开头）、命令（如果单词是命令行的第一个单词）和主机名（如果单词以@开头）起作用。主机名的自动补齐只对/etc/hosts 目录下的主机名生效。

有一些控制和元键序列与自动补齐功能相关联（见表 8-4）。

表 8-4　自动补齐命令

组合键	作用
Alt-$	显示所有可能的自动补齐列表。在大多数系统中，可通过按两次 Tab 键实现，而且也会更容易一些
Alt-*	插入所有可能的匹配项。当需要用到一个以上的匹配项时，将比较有用

除了以上这些，还有相当多的组合键，可以在 bash man 页面的 READLINE 部分获取更多的相关内容列表。

可编程的自动补齐

bash 的当前版本提供了一种称为“可编程的自动补齐”的工具。可编程自动补齐允许用户（更可能是发行版本提供商）添加附加的自动补齐规则。一般来说，这样做是为了支持特定的应用。例如，可以为一个命令的可选列表，或者是为了匹配某种应用支持的特定的文件类型，而添加自动补齐。默认情况下，Ubuntu 定义了一个相当大的规则集合。可编程自动补齐通过 shell 函数来实现的，shell 函数是一种小型 shell 脚本，这个将在后面的章节介绍。如果你好奇的话，试一下：

```
set | less
```

看看是否可以找到它们。默认情况下，并不是所有的发行版本都包含它们。

8.3 使用历史命令

第 1 章我们已经提到，bash 会保存使用过命令的历史记录。这些命令的历史记录列表保存在用户主目录的.bash_history 文件中。这些历史记录非常有用，可以大大减少用户敲打键盘的次数，特别是和命令行编辑结合使用的时候。

8.3.1 搜索历史命令

任何情况下，我们都可以通过如下命令查看历史记录的内容列表。

```
[me@linuxbox ~]$ history | less
```

bash 默认会保存用户最近使用过的 500 个命令。其中，500 是个默认值，关于如何改变这个默认值将在第 11 章介绍。假设我们想找到用来列出/usr/bin 目录下内容的命令，我们可以这样做：

```
[me@linuxbox ~]$ history | grep /usr/bin
```

假设得到的搜索结果中有一行包含如下有趣的命令。

```
88 ls -l /usr/bin > ls-output.txt
```

数字 88 表示这个命令行在历史记录列表中所处的行号，我们可以通过使用

名为历史记录扩展（history expansion）的扩展类型来立即使用它。为了使用我们发现的命令行，可以如下操作：

```
[me@linuxbox ~]$ !88
```

bash 将把!88 扩展为历史列表中第 88 行的内容。稍后将介绍历史记录扩展的其他形式。

bash 也支持以递增方式搜索历史记录。也就是说，当搜索历史记录时，随着输入字符数的增加，bash 会相应地改变搜索范围。按下 Ctrl-R 键，接着输入你要查找的内容，可以开始递增式的搜索。当找到要查找的内容时，按 Enter 键表示执行此命令，而按 Ctrl-J 将把搜索到的内容从历史记录列表中复制到当前命令行。当要查找下一个匹配项时（即向前搜索历史记录），再次按下 Ctrl-R 键。若要退出搜索，按下 Ctrl-G 或者 Ctrl-C 即可。请看下面的例子。

```
[me@linuxbox ~]$
```

首先按下 Ctrl-R。

```
(reverse-i-search) '':
```

提示符发生改变，提示正在进行逆向递增式搜索。称为“逆向”是因为查找的是从“现在”到过去的某个时间之间的操作。接下来，输入要查找的内容，这个例子中是查找/usr/bin。

```
(reverse-i-search) '/usr/bin': ls -l /usr/bin > ls-output.txt
```

很快搜索操作返回了结果。此时我们可按 Enter 键执行搜索结果，也可按下 Ctrl-J 把搜索结果复制到当前命令行以便作进一步的编辑。假定按下 Ctrl-J，把搜索结果复制到当前命令行。

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

shell 将实时响应，命令行将被加载，准备运行。

表 8-5 列出了一些用来手动操作历史记录的组合键。

表 8-5 历史记录命令

组合键	作用
Ctrl-P	移动到前一条历史记录。相当于向上箭头键
Ctrl-N	移动到后一条历史记录。相当于向下箭头键
Alt-<	移动到历史记录列表的开始处
Alt->	移动到历史记录列表的结尾处。即当前命令行

续表

组合键	作用
Ctrl-R	逆向递增地搜索。从当前命令行向前递增搜索
Alt-P	逆向非递增地搜索。按下这个组合键，接着输入待搜索的字符串，在按 Enter 键后，搜索才真正开始执行
Alt-N	向前非递增地搜索
Ctrl-O	执行历史记录列表中的当前项，执行完跳到下一项。若要把历史记录中的一系列命令重新执行一遍，使用该组合键将很方便

8.3.2 历史记录扩展

shell 提供了一种专门用来扩展历史记录项的方式——使用!字符。前面我们曾提到过如何通过在感叹号后面跟数字的方式，将来自历史记录列表中的命令插入到命令行中。除了这种方式，还有很多其他的扩展特性（见表 8-6）。

当使用“！ string”和“！？ string”时，请务必小心谨慎，除非对历史记录中的内容非常确信。

历史记录扩展机制中还有很多其他的可用特点，但是该主题太过晦涩难懂，此处不再讨论。你可以查阅 bash 帮助页面中的“HISTORY EXPANSION”部分获取更多细节。

表 8-6 历史记录扩展命令

序列	行为
!!	重复最后一个执行的命令。按向上箭头键再按 Enter 键也可实现相同的功能，而且操作更简单
!number	重复历史记录中第 number 行的命令
! string	重复最近的以 string 开头的历史记录
!?string	重复最近的包含 string 的历史记录

脚本

除了 bash 中的命令历史特性外，大部分 Linux 发行版本都包含一个称为脚本（script）的程序，它记录了 shell 的整个会话，并且将会话保存到一个文件中。该命令的基本语法是：

```
script [file]
```

其中 file 为用来保存会话记录的文件名。如果没有指定文件，默认使用文件 typescript。脚本（script）的 man 页面给出了该程序的所有可选项和特性。

8.4 本章结尾语

本章介绍了 shell 提供的一些键盘操作技巧，它们能够帮助打字员减少工作量。随着时间的推移，你会越来越多地接触到命令行，到时候你会翻阅这一章的内容，以获得更多的键盘使用技巧。当前，只需将它们当做一个虽然有用但是当前没有必要掌握的可选技能即可。

第 9 章 权限

传统的 UNIX 操作系统与那些传统的 MS-DOS 操作系统不同，区别在于它们不仅是多重任务处理（multitasking）系统，而且还是多用户（multiuser）系统。

确切地说，这意味着什么呢？这意味着同一时间内可以有多个用户使用同一台计算机。虽然一台标准的计算机可能只包含一个键盘和一台显示器，但是它仍然可以同时被一个以上的用户使用。例如，如果计算机连接到一个网络或者互联网中，远程用户可以通过 ssh（安全 shell）登录并且操作这台计算机。事实上，远程用户可以执行图形化应用程序，而且图形化的输出结果将会出现在远程显示器上。X 窗口系统把这个作为基本设计理念的一部分，并支持这种功能。

Linux 的多用户功能并不是最近的“创新”，而是深嵌在操作系统设计理念中的一个特色功能。想想 UNIX 系统诞生时的背景环境，该功能的出现有着重大的意义。很多年前，在计算机“个人化”之前，计算机普遍体积大，价格昂

贵，而且都是集中控制的。例如，一个典型的校园计算机系统，是由一台放置在某建筑物中的大型中央计算机以及遍布校园的各台终端机组成的，每台终端机都连接到中央计算机上。这台中央计算机可以同时支持很多用户。

为了保证多用户功能实际可用，系统特别设计了一种方案来保护当前用户不受其他用户操作的影响。毕竟，一个用户的操作不能导致计算机崩溃，一个用户的操作界面也不能显示属于另一个用户的文件。

本章将介绍系统安全的基础知识以及如下命令的使用。

- id：显示用户身份标识。
- chmod：更改文件的模式。
- umask：设置文件的默认权限。
- su：以另一个用户的身份运行 shell。
- sudo：以另一个用户的身份来执行命令。
- chown：更改文件所有者。
- chgrp：更改文件所属群组。
- passwd：更改用户密码。

9.1 所有者、组成员和其他所有用户

我们在第 4 章讲解文件系统时，当试图查看类似/etc/shadow 的文件时，会遇到下面的问题。

```
[me@linuxbox ~]$ file /etc/shadow
/etc/shadow: regular file, no read permission
[me@linuxbox ~]$ less /etc/shadow
/etc/shadow: Permission denied
```

产生这种错误信息的原因是，作为一个普通用户，没有读取这个文件的权限。

在 UNIX 安全模型中，一个用户可以拥有（own）文件和目录。当一个用户拥有一个文件或者目录时，它将对该文件或目录的访问权限拥有控制权。反过来，用户又归属于一个群组（group），该群组由一个或者多个用户组成，组中用户对文件和目录的访问权限由其所有者授予。除了可以授予群组访问权限之外，文件所有者也可以授予所有用户一些访问权限，在 UNIX 术语中，所有用户是指整个世界（world）。使用 id 命令可以获得用户身份标识的相关信息，如

下所示。

```
[me@linuxbox ~]$ id
uid=500(me) gid=500(me) groups=500(me)
```

查看 id 命令的输出结果。在创建用户账户的时候，用户将被分配一个称为用户 ID（user ID）或者 uid 的号码。为了符合人们的使用习惯，用户 ID 与用户名一一映射。同时用户将被分配一个有效组 ID（primary group ID）或者称为 gid，而且该用户也可以归属于其他的群组。前面的例子是在 Fedora 系统中运行的结果。在其他系统中，比如 Ubuntu 系统，输出结果可能会有一些不同。

```
[me@linuxbox ~]$ id
uid=1000(me) gid=1000(me)
groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(
plugdev),108(lpadmin),114(admin),1000(me)
```

我们可以发现，两个系统中用户的 uid 和 gid 号码是不同的。原因很简单，因为在 Fedora 系统中，普通用户账户是从 500 开始编号的，而在 Ubuntu 系统中则是从 1000 开始编号。同时我们也可以发现，Ubuntu 系统中的用户归属于更多的群组。这和 Ubuntu 系统管理系统设备和服务权限的方式有关。

那么这些信息从何而来呢？类似于 Linux 系统中的很多情况，这些信息来源于一系列的文本文件。用户账户定义在文件/etc/passwd 中，用户组定义在文件/etc/group 文件中。在创建用户账户和群组时，这些文件随着文件/etc/shadow 的变动而修改，文件/etc/shadow 中保存了用户的密码信息。对于每一个用户账户，文件/etc/passwd 中都定义了对应用户的用户（登录）名、uid、gid、账户的真实姓名、主目录以及登录 shell 信息。如果查看文件/etc/passwd 和文件/etc/group 的内容，那么你将会发现除了普通用户账户信息之外，文件中还有对应于超级用户（uid 为 0）和其他不同种类的系统用户的账户信息。

在第 10 章中我们介绍进程时，你将会发现这些其他的“用户”中有一些实际上是相当忙碌的。

许多类 UNIX 系统会把普通用户分配到一个公共的群组中（比如，users），然而现在的 Linux 操作则是创建一个独一无二的，只有一个用户的群组，而且组名和用户的名字相同。这使得特定类型的权限分配变得更加容易。

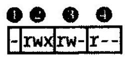
9.2　读取、写入和执行

对文件和目录的访问权限是按照读访问、写访问以及执行访问来定义的。

当我们查看 ls 命令的输出结果时，可以得到一些线索，了解其实现方式。

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1   me   me    0 2012-03-06 14:52 foo.txt
```

列在输出结果中的前 10 个字符表示的是文件属性（file attribute，见图 9-1）。其中第一个字符表示文件类型（file type）。表 9-1 列出了最可能见到的文件类型（还有其他的不常见类型）。



❶ 文件类型（见表9-1）

❷ 所有者权限（见表9-2）

❸ 组权限（见表9-2）

❹ 其他用户权限（见表9-2）

图 9-1 文件属性的分类

表 9-1 文件类型

属性	文件类型
-	普通文件
d	目录文件
l	符号链接。注意对于符号链接文件，剩下的文件属性始终是 rwxrwxrwx，它是个伪属性值。符号链接指向的文件的属性才是真正的文件属性
c	字符设备文件。该文件类型表示以字节流形式处理数据的设备，如终端或调制解调器
b	块设备文件。该文件类型表示以数据块方式处理数据的设备，如硬盘驱动或者光盘驱动

文件属性中剩下的 9 个字符称为文件模式（file mode），分别表示文件所有者、文件所属群组以及其他所有用户对该文件的读取、写入和执行权限。

分别设置 r、w 和 x 的模式属性将会对文件和目录带来不同的影响，如表 9-2 所示。

表 9-2 权限属性

属性	文件	目录
r	允许打开和读取文件	如果设置了执行权限，那么允许列出目录下的内容
w	允许写入或者截短文件；如果也设置了执行权限，那么目录中的文件允许被创建、被删除以及被重命名	但是该权限不允许重命名或者删除文件。是否能重命名和删除文件由目录权限决定

续表

属性	文件	目录
x	允许把文件当作程序一样来执行。用脚本语言写的程序文件必须被设置为可读，以便能被执行	允许进入目录下，例如 cd directory

表 9-3 给出了一些文件属性设置的例子。

表 9-3　权限属性实例

文件属性	含义
-rwx------	普通文件，文件所有者具有读取、写入和执行权限。组成员和其他所有用户都没有任何访问权限
-rw-------	普通文件，文件所有者具有读取和写入权限。组成员和其他所有用户都没有任何访问权限
-rw-r—r--	普通文件，文件所有者具有读取和写入权限。文件所有者所在群组的成员可以读取该文件。该文件对于所有用户来说都是可读的
-rwxr-xr-x	普通文件，文件所有者具有读取、写入和执行权限。其他所有用户也可以读取和执行该文件
-rw-rw----	普通文件，只有文件所有者和文件所有者所在群组的成员具有读取和执行权限
Lrwxrwxrwx	符号链接。所有的符号链接文件显示的都是“伪”权限属性，真正的权限属性由符号链接指向的实际文件决定
dwxrwx---	目录文件。文件所有者和所有者所在群组的成员都可以进入该目录，而且可以创建、重命名和删除该目录下的文件
drwxr-x---	目录文件。文件所有者可以进入该目录，而且可以创建、重命名和删除该目录下的文件。所有者所在群组的成员可以进入该目录，但是不能创建、重命名和删除该目录下的文件

9.2.1 chmod——更改文件模式

我们可以使用 chmod 命令来更改文件或者目录的模式（权限）。需要注意的是只有文件所有者和超级用户才可以更改文件或者目录的模式。chmod 命令支持两种不同的改变文件模式的方式——八进制数字表示法和符号表示法。首先我们来学习八进制数字表示法。

八进制数字表示法

八进制表示法指的是使用八进制数字来设置所期望的权限模式。因为每个八进制数字对应着 3 个二进制数字，所以这种对应关系正好可以和用来存储文件模式的结构方式一一映射。表 9-4 形象地说明了这一点。

表 9-4 以二进制和八进制方式表示文件模式

八进制	二进制	文件模式
0	000	---
1	001	--x
2	010	-w-
3	011	-wx
4	100	r--
5	101	r-x
6	110	rw-
7	111	rwx

八进制到底是什么

八进制（octal，以 8 为基数）和十六进制（hexadecimal，以 16 为基数）都是数字系统，它们经常用来表示计算机中的数字。由于人生来就有十个手指（至少大多数人都是如此），所以采用以 10 为基数的数字系统来计数。而另一方面，计算机生来只有一个手指，因此它们采用二进制（以 2 为基数）来完成所有的计数。它们的数字系统只有两个数字：0 和 1。所以在二进制中，以这种方式计数：0、1、10、11、100、101、110、111、1000、1001、1010、1011...

八进制使用数字 0~7 来计数，即 0、1、2、3、4、5、6、7、10、11、12、13、14、15、16、17、20、21...

十六进制使用数字 0~9，加上字母 A~F 来计数，即 0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F、10、11、12、13...

二进制出现的意义可以理解（由于计算机生来只有一个手指），但是八进制和十六进制又有什么用处呢？答案是为了给人们提供方便。许多时候，计算机中的小部分数据以位模式（bit pattern）来表示。以 RGB 颜色为例子来说明。在大多数的计算机显示器中，每个像素由三种颜色组件组成：8 位红色、8 位绿色以及 8 位蓝色。一种漂亮的中蓝色由一组 24 位数字来表示，即 010000110110111111001101。

没有人愿意整天都读写这种类型的数字。这里使用另一种数字系统将更简单。十六进制中的一个数字代表二进制中的四个数字。八进制中的一个数字代表二进制中的三个数字。因此，这 24 位中蓝色二进制数将可以压缩成一个 6 位十六进制数：436FCD。由于十六进制的数字和二进制的位“排列整齐”，所以该颜色中的红色对应 43，绿色对应 6F，蓝色对应 CD。

目前十六进制（经常称为 hex）比八进制的使用更普遍，但是八进制数字

> 可以用来表示 3 位二进制数的功能是非常有用的，接下来我们将很快将可以看到这一点。

通过使用 3 位八进制数字，我们可以分别设置文件所有者、组成员和其他所有用户（world）的文件模式。

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me    me 0 2012-03-06 14:52 foo.txt
[me@linuxbox ~]$ chmod 600 foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw------- 1 me    me 0 2012-03-06 14:52 foo.txt
```

通过传递参数 600，我们可以设置文件所有者具有读写权限，而取消组用户和其他所有用户（world）的所有权限。虽然看起来，要记住八进制和二进制之间的映射关系好象不是那么简单，但是实际上，常用的也就只有这几个而已：7（rwx）、6（rw-）、5（r-x）、4（r--）和 0（---）。

符号表示法

chmod 命令支持一种符号表示法来指定文件模式。该符号表示法分为三部分：更改会影响谁、要执行哪个操作以及要设置哪种权限。可以通过字符 u、g、o 和 a 的组合来指定要影响的对象，如表 9-5 所示。

表 9-5　chmod 命令符号表示法

符号	含义
u	user 的简写，表示文件或者目录的所有者
g	文件所属群组
o	others 的简写，表示其他所有用户
a	all 的简写，是‘u’,‘g’和‘o’三者的组合

如果没有指定字符，则假定使用 all。操作符“+”表示添加一种权限，“-”表示删除一种权限，“=”表示只有指定的权限可用，其他所有的权限都被删除。

权限由字符“r”、“w”和“x”来指定。表 9-6 列出了一些符号表示法的实例。

表 9-6　chmod 命令符号表示法实例

符号	含义
u+x	为文件所有者添加可执行权限
u-x	删除文件所有者的可执行权限
+x	为文件所有者、所属群组和其他所有用户添加可执行权限，等价于 a+x

续表

符号	含义
o-rw	除了文件所有者和所属群组之外，删除其他所有用户的读写权限
go=rw	除了文件所有者之外，设置所属群组和其他所有用户具有读写权限。如果所属群组或者其他所有用户之前已经具有可执行权限，那么删除他们的可执行权限
u+x，go=rx	为文件所有者添加可执行权限，同时设置所属群组和其他所有用户具有读权限和可执行权限。指定多种权限时，需用逗号分隔

有些人喜欢使用八进制表示法，而有些人则真的很喜欢用符号表示法。符号表示法的优点在于允许设置单个属性，而不影响其他的任何属性。

我们可以查看 chmod 命令的帮助页面，以获取更多的细节内容和选项信息。关于--recursive 选项，我们需要注意，它对文件和目录都起作用，所以该选项并不如想象中的那么有用，因为用户很少会想要给文件和目录设置相同的权限。

9.2.2 采用 GUI 设置文件模式

现在，我们已经知道了如何设置文件和目录的权限，这样就可以更好地理解 GUI 中的设置权限对话框了。在 Nautilus（GNOME 桌面系统）和 Konqueror（KDE 桌面系统）中，右击文件或者目录图标都将会弹出一个属性对话框。图 9-2 是 KDE 3.5 环境下运行的一个例子。

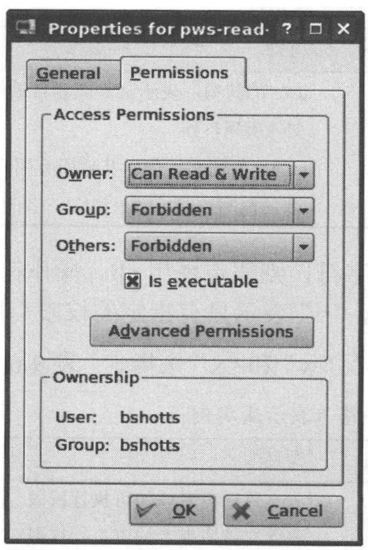


图 9-2 KDE 3.5 运行环境下文件属性对话框

可以看到，这个对话框中可以设置文件所有者、用户组和其他所有用户的权限。在 KDE 运行环境下，单击该对话框中的“Advanced Permissions（高级权限）”按钮，将弹出另一个对话框，在这个对话框中允许单独设置各个模式属性。另一种易于理解的实现方式就是使用命令行！

9.2.3 umask——设置默认权限

umask 命令控制着创建文件时指定给文件的默认权限。它使用八进制表示法来表示从文件模式属性中删除一个位掩码。

参见下面的例子：

```
[me@linuxbox ~]$ rm -f foo.txt
[me@linuxbox ~]$ umask
0002
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me   me    0 2012-03-06 14:53 foo.txt
```

首先，删除 foo.txt 文件存在的所有副本，以保证一切都是重新开始。下一步，运行不带任何参数的 umask 命令，查看当前掩码值，得到的值是 0002（0022 是另一个常用默认值），它是掩码的八进制表示形式。接着创建文件 foo.txt 的一个新实例，查看该文件的权限。

可以发现，文件所有者和组都获得了读写权限，而其他所有用户则只获得读权限。其他所有用户没有写权限的原因在于掩码值。重复执行该实例，不过这次是自己设置掩码值。

```
[me@linuxbox ~]$ rm foo.txt
[me@linuxbox ~]$ umask 0000
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-rw- 1 me   me    0 2012-03-06 14:58 foo.txt
```

在设置掩码为 0000（实际上是关闭该功能）时，可以看到其他所有用户也拥有写权限了。为了理解它是如何实现的，再来看看八进制数。如果把该掩码展开成二进制形式，然后再与属性进行对比，那么就能明白是怎么回事了。

原始文件模式	---	rw-	rw-	rw-
掩码	000	000	000	010
结果	---	rw-	rw-	r--

先忽略掉掩码中前面的 0（稍后再看），观察掩码中出现 1 的地方，将会发

现 1 的位置对应的属性被删除——在这个例子中对应的是其他所有用户的写权限。这就是掩码的操作方式。掩码的二进制数值中每个出现 1 的位置，其对应的属性都被取消。如果设置掩码值为 0022，那么具体操作如下。

原始文件模式	---	rw-	rw-	rw-
掩码	000	000	000	010
结果	---	rw-	rw-	r--

同样地，二进制数值中 1 出现的位置，其对应的属性都被取消。再试一下其他的掩码值（尝试一些带数字 7 的），以熟悉掩码的操作方式。记得每次操作完之后清理文件，并把掩码值还原到默认值。

```
[me@linuxbox ~]$ rm foo.txt; umask 0002
```

大多数情况下，你并不需要修改掩码值，系统提供的默认掩码值就很好了。然而，在一些高安全级别的环境下，你则需要控制掩码值。

一些特殊权限

虽然通常看到的八进制权限掩码都是用三位数字表示的，但是，确切地说，从技术层面上来看，它是用四位数字来表示的。为什么呢？因为，除了读取、写入和执行权限之外，还有一些其他的较少用到的权限设置。

其中之一是 setuid 位（八进制表示为 4000）。当把它应用到一个可执行文件时，有效用户 ID 将从实际用户 ID（实际运行该程序的用户）设置成该程序所有者的 ID。大多数情况下，该权限设置通常应用于一些由超级用户所拥有的程序。当普通用户运行一个具有“setuid root”（已设置 setuid 位，由 root 用户所有）属性的程序时，该程序将以超级用户的权限来执行。这使得该程序可以访问一些普通用户通常禁止访问的文件和目录。很明显，这会带来安全方面的问题，因此允许设置 setuid 位的程序个数必须控制在绝对小的范围内。

第二个是 setgid 位（八进制表示为 2000）。类似于 setuid 位，它会把有效组 ID 从该用户的实际组 ID 更改为该文件所有者的组 ID。如果对一个目录设置 setgid 位，那么在该目录下新创建的文件将由该目录所在组所有，而不是由文件创建者所在组所有。当一个公共组下的成员需要访问共享目录下的所有文件时，设置 setgid 位将很有用，并不需要关注文件所有者所在的有效组。

第三个是 sticky 位（八进制表示为 1000）。它是从传统 UNIX 中继承下来的，可以标记一个可执行文件为“不可交换的”。在 Linux 中，会忽略文件的

sticky 位，但是如果对一个目录设置 sticky 位，那么将能阻止用户删除或者重命名文件，除非用户是这个目录的所有者、文件所有者或者是超级用户。它常用来控制对共享目录（比如，/tmp）的访问。

这里有一些使用 chmod 命令和符号表示法来设置这些特殊权限的实例。首先，授予程序 setuid 权限：

```
chmod u+s 程序名
```

下一步，授予目录 setgid 权限：

```
chmod g+s 目录
```

最后，授予目录 sticky 位权限：

```
chmod +t 目录
```

使用 ls 命令可以查看这些特殊权限的设置结果。首先，设置了 setuid 位的程序：

```
-rwsr-xr-x
```

具有 setgid 属性的目录：

```
drwxrwsr-x
```

设置了 sticky 位的目录：

```
drwxrwxrwt
```

9.3 更改身份

在很多时候，我们会发现可以拥有另一个用户的身份是很有必要的。我们经常会需要获得超级用户的特权来执行一些管理任务，但是也可以“变成”另一个普通用户来执行这些任务，就好像是在测试一个账户。有三种方法用来转换身份，具体如下。

- 注销系统并以其他用户的身份重新登录系统。
- 使用 su 命令。
- 使用 sudo 命令。

因为大家都知道如何操作第一种方法，而且它不如其他两种方法来得方便，所以这里跳过第一种方法。在 shell 会话状态下，使用 su 命令将允许你假定为另一个用户的身份，既可以以这个用户的 ID 来启动一个新的 shell 会话，也可以以这个

用户的身份来发布一个命令。使用 sudo 命令将允许管理者创建一个称为/etc/sudoer 的配置文件，并且定义一些特定的命令，这些命令只有被赋予为假定身份的特定用户才允许执行。选择使用哪个命令在很大程度上取决于使用的 Linux 发行版本。有些发行版本可能对两个命令都支持，但是它的系统配置可能只是偏向于其中一个。首先我们来介绍 su 命令。

9.3.1 su——以其他用户和组 ID 的身份来运行 shell

su 命令用来以另一个用户的身份来启动 shell。该命令的一般形式如下。

```
su [-[l]] [user]
```

如果包含“-l”选项，那么得到的 shell 会话界面将是用于指定用户的登录 shell（login shell）界面。这就意味着，该指定用户的运行环境将被加载，而且其工作目录也将更改为该指定用户的主目录。这也常常我们是想要得到的结果。如果没有指定用户，那么默认假定为超级用户。需要注意的是，-l 可以缩写为-，而且这一形式经常被使用。我们可以通过以下的操作来以超级用户的身份启动 shell。

```
[me@linuxbox ~]$ su -
Password:
[root@linuxbox ~]#
```

在输入 su 命令后，系统会提示输入该超级用户的密码。如果密码输入正确，那么将会出现新的 shell 提示符，该提示符表示该 shell 将拥有超级用户的特权（提示符的末尾字符是#，而不是$），而且当前的工作目录现在也是用于超级用户的主目录（通常情况下为/root）。一旦进入了这个新的 shell 环境，我们就可以以超级用户的身份执行命令了。在使用结束时，输入 exit，将返回到之前的 shell 环境。

```
[root@linuxbox ~]# exit
[me@linuxbox ~]$
```

我们也可以使用 su 命令执行单个命令，而不需要开启一个新的交互式命令界面，操作方式如下。

```
Su –c 'command'
```

使用这种格式，单个命令行将被传递到一个新的 shell 环境下进行执行。这里需要用单引号把命令行引起来，这一点很重要。因为该命令扩展并不希望在当前 shell 环境下执行，而是希望在新的 shell 环境下执行。

```
[me@linuxbox ~]$ su -c 'ls -l /root/*'
Password:
-rw------- 1 root root      754 2011-08-11 03:19 /root/anaconda-ks.cfg

/root/Mail:
total 0
[me@linuxbox ~]$
```

9.3.2 sudo——以另一个用户的身份执行命令

sudo 命令在很多方面都类似于 su 命令，但是它另外还有一些重要的功能。管理者可以通过配置 sudo 命令，使系统以一种可控的方式，允许一个普通用户以一个不同的用户身份（通常是超级用户）执行命令。在特定情况下，用户可能被限制为只能执行一条或者几条特定的命令，而对其他命令没有执行权限。另一个重要的区别在于，使用 sudo 命令并不需要输入超级用户的密码。使用 sudo 命令时，用户只需要输入自己的密码来进行认证。比如说，配置 sudo 命令来允许普通用户运行一个虚构的备份程序（称为 backup_script），这个程序需要超级用户的权限。

通过 sudo 命令，该程序将会以如下的方式运行。

```
[me@linuxbox ~]$ sudo backup_script
Password:
System Backup Starting...
```

在输入 sudo 命令后，系统将提示输入用户自己的密码（而不是超级用户的密码），而且一旦认证通过，指定的命令就将被执行。su 命令和 sudo 命令之间的一个重要区别在于 sudo 命令并不需要启动一个新的 shell 环境，而且也不需要加载另一个用户的运行环境。这意味着，使用 sudo 命令的时候并不需要用单引号把命令行引起来。需要注意的是，我们可以通过指定不同的选项来改变命令执行的效果。查看 sudo 命令的帮助页面可以获得更多的细节内容。

要想知道 sudo 命令可以授予哪些权限，可以使用-l 选项来查看，具体如下。

```
[me@linuxbox ~]$ sudo -l
User me may run the following commands on this host:
     (ALL) ALL
```

Ubuntu 与 sudo

普通用户经常会遇到这样的一个问题，即如何完成某些特定的、需要超级用户权限才能完成的任务。这些任务包括安装和更新软件、编辑系统配置文件和访问设备等。在 Windows 世界中，这些任务通常是通过授予用户管理

员权限来完成的。然而，这也会使得用户执行的程序具有相同的功能。大多数情况下，这正是用户所期望的结果，但是这样也会使得类似病毒这样的恶意软件（malware）可以随意地操作计算机。

在 UNIX 世界中，由于 UNIX 多用户的传统特性，普通用户和管理者之间一直都存在着更大的差别。UNIX 采用的方法是只有在需要的时候才允许授予超级用户的特权，通常使用 su 命令和 sudo 命令来实现这一操作。

几年前，大多数的 Linux 发布版本都依靠 su 命令来达成这一目的。su 命令并不需要 sudo 命令所要求的配置环境，而且按照 UNIX 的传统，su 命令会拥有一个 root 账户。但是，这将会产生一个问题，即用户在不是很必要的情况下也会试图以 root 用户的身份来进行操作。事实上，一些用户专门以 root 用户的身份来操作系统，从而摆脱那些烦人的“permission denied（权限被拒绝）”信息，这就使得 Linux 系统的安全级别降低到和 Windows 系统的安全级别一样。因此，使用 su 命令并不是一个好主意。

在推出 Ubuntu 的时候，Ubuntu 的创造者采取了一个不同的策略。默认情况下，Ubuntu 不允许用户以 root 账户的身份登录（因为不能成功为 root 账户设置密码），取而代之的是使用 sudo 命令来授予超级用户的特权。最初的用户账户可以通过 sudo 命令来获得超级用户的全部权限，后面的用户账户也可以被授予相似的权限。

9.3.3 chown——更改文件所有者和所属群组

chown 命令用来更改文件或者目录的所有者和所属群组。使用这个命令需要超级用户的权限。chown 命令的语法格式如下。

```
chown [ower][:[group]] file ...
```

chown 命令更改的是文件所有者还是文件所属群组，或者对两者都更改，取决于该命令的第一个参数。表 9-7 列出了一些实例。

表 9-7 chown 命令参数实例

参数	结果
bob	把文件所有者从当前所有者更改为用户 bob
bob:users	把文件所有者从当前所有者更改为用户 bob，并把文件所属群组更改为 users 组
:admins	把文件所属群组更改为 admins 组，文件所有者不变
bob:	把文件所有者从当前所有者更改为用户 bob，并把文件所属群组更改为用户 bob 登录系统时所属的组

假设有两个用户——拥有超级用户权限的 janet 和没有该权限的 tony。用户 janet 想要从她的主目录复制一个文件到用户 tony 的主目录中。因为用户 janet 希望 tony 能够编辑该文件，所以 janet 把该文件的所有者从 janet 更改为 tony，具体操作如下。

```
[janet@linuxbox ~]$ sudo cp myfile.txt ~tony
Password:
[janet@linuxbox ~]$ sudo ls -l ~tony/myfile.txt
-rw-r--r-- 1 root root 8031 2012-03-20 14:30 /home/tony/myfile.txt
[janet@linuxbox ~]$ sudo chown tony: ~tony/myfile.txt
[janet@linuxbox ~]$ sudo ls -l ~tony/myfile.txt
-rw-r--r-- 1 tony tony 8031 2012-03-20 14:30 /home/tony/myfile.txt
```

这里我们可以看到，用户 janet 首先把文件从她的目录复制到用户 tony 的主目录中。接下来，janet 把该文件的所有者从 root（使用 sudo 命令的结果）更改为 tony。通过在第一个参数末尾加上冒号，janet 也把文件的所属群组更改成了 tony 登录系统时所属的组，该组名碰巧也叫 tony。

值得注意的是，在 janet 第一次使用了 sudo 命令之后，系统为什么没有提示她输入她的密码呢？这是因为，在大多数的配置环境下，sudo 命令会“信任”用户几分钟（直到计时结束）。

9.3.4 chgrp——更改文件所属群组

在更早的 UNIX 版本中，chown 命令只能更改文件的所有者，而不能改变文件所属群组。为了达到这个目的，我们可以使用一个独立的命令 chgrp 来实现。该命令除了限制多一点之外，和 chown 命令的使用方式几乎相同。

9.4 权限的使用

既然我们已经学习了权限是如何工作的，那么现在是时候学以致用了。接下来让我们看一个常见问题的解决方案——创建一个共享目录。假设有两个用户，分别命名为 bill 和 karen。他们都有音乐 CD 集，并想要创建一个共享目录，在该目录下他们各自以 Ogg Vorbis 格式或者 MP3 格式来存储音乐文件。用户 bill 通过 sudo 命令获得了超级用户的访问权限。

第一件需要做的事情，就是创建一个以 bill 和 karen 为成员的组。通过使用 GNOME 的图形化用户管理工具，bill 创建了一个组，命名为 music，并且把用户 bill 和 karen 添加到该组中，如图 9-3 所示。

员权限来完成的。然而，这也会使得用户执行的程序具有相同的功能。大多数情况下，这正是用户所期望的结果，但是这样也会使得类似病毒这样的恶意软件（malware）可以随意地操作计算机。

在 UNIX 世界中，由于 UNIX 多用户的传统特性，普通用户和管理者之间一直都存在着更大的差别。UNIX 采用的方法是只有在需要的时候才允许授予超级用户的特权，通常使用 su 命令和 sudo 命令来实现这一操作。

几年前，大多数的 Linux 发布版本都依靠 su 命令来达成这一目的。su 命令并不需要 sudo 命令所要求的配置环境，而且按照 UNIX 的传统，su 命令会拥有一个 root 账户。但是，这将会产生一个问题，即用户在不是很必要的情况下也会试图以 root 用户的身份来进行操作。事实上，一些用户专门以 root 用户的身份来操作系统，从而摆脱那些烦人的"permission denied（权限被拒绝）"信息，这就使得 Linux 系统的安全级别降低到和 Windows 系统的安全级别一样。因此，使用 su 命令并不是一个好主意。

在推出 Ubuntu 的时候，Ubuntu 的创造者采取了一个不同的策略。默认情况下，Ubuntu 不允许用户以 root 账户的身份登录（因为不能成功为 root 账户设置密码），取而代之的是使用 sudo 命令来授予超级用户的特权。最初的用户账户可以通过 sudo 命令来获得超级用户的全部权限，后面的用户账户也可以被授予相似的权限。

9.3.3 chown——更改文件所有者和所属群组

chown 命令用来更改文件或者目录的所有者和所属群组。使用这个命令需要超级用户的权限。chown 命令的语法格式如下。

```
chown [ower][:[group]] file ...
```

chown 命令更改的是文件所有者还是文件所属群组，或者对两者都更改，取决于该命令的第一个参数。表 9-7 列出了一些实例。

表 9-7 chown 命令参数实例

参数	结果
bob	把文件所有者从当前所有者更改为用户 bob
bob:users	把文件所有者从当前所有者更改为用户 bob，并把文件所属群组更改为 users 组
:admins	把文件所属群组更改为 admins 组，文件所有者不变
bob:	把文件所有者从当前所有者更改为用户 bob，并把文件所属群组更改为用户 bob 登录系统时所属的组

```
[bill@linuxbox ~]$ > /usr/local/share/Music/test_file
[bill@linuxbox ~]$ ls -l /usr/local/share/Music
-rw-r--r-- 1 bill bill 0 2012-03-24 20:03 test_file
```

实际上，这会产生两个问题。第一个问题是，系统中的默认掩码是 0022，这将不会允许组成员对属于组内其他成员的文件执行写入操作。如果共享目录中只包含文件，那么这倒不是问题，但是因为该目录下将存放音乐文件，而音乐文件一般都是按照艺术家和唱片集的层次结构来组织分类的，所以组成员需要拥有能在同组其他成员创建的目录下创建文件和目录的权限。需要把 bill 和 karen 使用的掩码值更改成 0002。

第二个问题是，由成员创建的每一个文件和目录都将被设置为归属于该用户的有效组，而不是归属于 music 组。可以通过对该目录设置 setgid 位来修复这个问题。

```
[bill@linuxbox ~]$ sudo chmod g+s /usr/local/share/Music
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxrwsr-x 2 root music 4096 2012-03-24 20:03 /usr/local/share/Music
```

现在来测试一下，看看是否新的权限修复了这个问题。bill 把他的掩码值设置为 0002，删除了之前的测试文件，又创建了一个新的测试文件和目录。

```
[bill@linuxbox ~]$ umask 0002
[bill@linuxbox ~]$ rm /usr/local/share/Music/test_file
[bill@linuxbox ~]$ > /usr/local/share/Music/test_file
[bill@linuxbox ~]$ mkdir /usr/local/share/Music/test_dir
[bill@linuxbox ~]$ ls -l /usr/local/share/Music
drwxrwsr-x 2 bill   music 4096 2012-03-24 20:24 test_dir
-rw-rw-r-- 1 bill   music 0 2012-03-24 20:22 test_file
[bill@linuxbox ~]$
```

当前创建的文件和目录都具有正确的权限，允许 music 组内的所有成员在 Music 目录下创建文件和目录。

最后剩下的一个问题是关于 umask 命令的。umask 命令设置的掩码值只能在当前 shell 会话中生效，在当前 shell 会话结束后，则必须重新设置。在第 11 章中，我们将介绍如何使得对 umask 命令掩码值的更改永久生效。

9.5　更改用户密码

本章的最后一个主题就是用户如何为自己设置密码（如果拥有超级用户权限，那么也可以为其他的用户设置密码）。使用 passwd 命令，可以设置或者更改密码。该命令的语法格式如下。

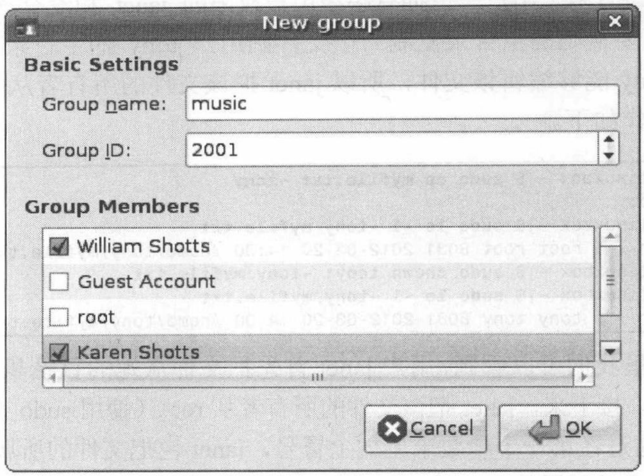


图 9-3 使用 GNOME 创建一个新组

接下来，bill 创建了存储音乐文件的目录。

```
[bill@linuxbox ~]$ sudo mkdir /usr/local/share/Music
Password:
```

因为 bill 正在操作的对象是他的主目录之外的文件，所以他需要拥有超级用户的权限。新创建的目录具有以下的所有权和权限。

```
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxr-xr-x 2 root root 4096 2012-03-21 18:05 /usr/local/share/Music
```

可以看到，这个目录由 root 用户所有，而且权限值为 755。要使得该目录可共享，bill 需要更改该目录的所属群组，而且该组需要具有写入权限。

```
[bill@linuxbox ~]$ sudo chown :music /usr/local/share/Music
[bill@linuxbox ~]$ sudo chmod 775 /usr/local/share/Music
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxrwxr-x 2 root music 4096 2012-03-21 18:05 /usr/local/share/Music
```

做所有的这些事情有什么意义呢？它意味着当前我们已经有了/usr/local/share/Music 目录，该目录由 root 用户所有，而且 music 组拥有该目录的读写权限。music 组的成员为 bill 和 karen，因此 bill 和 karen 都可以在目录/usr/local/share/Music 下创建文件。其他用户可以列出该目录下的内容，但是不能在该目录下创建文件。

但是我们仍然有一个问题。在当前的权限下，在 Music 目录下创建的文件和目录拥有用户 bill 和 karen 的常规权限。

第10章 进　程

现代操作系统通常都支持多重任务处理（multitasking）。多重任务处理是指系统通过快速切换运行中的程序来实现多任务的同时执行。Linux 内核通过使用进程来管理多重任务。进程是 Linux 用来安排不同程序等待 CPU 调度的一种组织方式。

有时候计算机运行速度会变得很慢，或者应用程序会停止响应。本章将介绍命令行中可用来查看程序当前运行情况以及终止运行异常的进程的一些工具。

本章将介绍以下命令。

- ps：显示当前所有进程的运行情况。
- top：实时显示当前所有任务的资源占用情况。
- jobs：列出所有活动作业的状态信息。
- bg：设置在后台中运行作业。

```
Passwd [user]
```

如果要更改的是用户自己的密码，那么只需要输入 passwd 命令。接下来 shell 将会提示用户输入旧密码和新密码。

```
[me@linuxbox ~]$ passwd
(current) UNIX password:
New UNIX password:
```

passwd 命令会试着强迫用户使用“强”密码。也就是说，它会拒绝接受太短的密码、与之前的密码相似的密码、字典中的单词作为密码或者是太容易猜到的密码。

```
[me@linuxbox ~]$ passwd
(current) UNIX password:
New UNIX password:
BAD PASSWORD: is too similar to the old one
New UNIX password:
BAD PASSWORD: it is WAY too short
New UNIX password:
BAD PASSWORD: it is based on a dictionary word
```

如果你具有超级用户的权限，那么可以通过指定一个用户名作为 passwd 命令的参数来为另一个用户设置密码。对于超级用户，还可以使用该命令的其他选项来设置账户锁定、密码失效等功能。你可以查看 passwd 命令的帮助页面获取更多的细节内容。

他字段信息。TTY 是 *teletype*（电传打字机）的缩写，代表了进程的控制终端（controlling terminal）。UNIX 在这里也显示了进程的运行时间，TIME 字段表示了进程消耗的 CPU 时间总和。可以看出，这两个进程都没有使计算机变得忙碌。

如果在 ps 命令后添加一个选项，那么我们将得到反映系统运行情况的更大视图界面，如下所示。

```
[me@linuxbox ~]$ ps x
  PID TTY        STAT   TIME COMMAND
 2799 ?          Ssl    0:00 /usr/libexec/bonobo-activation-server –ac
 2820 ?          Sl     0:01 /usr/libexec/evolution-data-server-1.10 --
15647 ?          Ss     0:00 /bin/sh /usr/bin/startkde
15751 ?          Ss     0:00 /usr/bin/ssh-agent /usr/bin/dbus-launch --
15754 ?          S      0:00 /usr/bin/dbus-launch --exit-with-session
15755 ?          Ss     0:01 /bin/dbus-daemon --fork --print-pid 4 –pr
15774 ?          Ss     0:02 /usr/bin/gpg-agent -s –daemon
15793 ?          S      0:00 start_kdeinit --new-startup +kcminit_start
15794 ?          Ss     0:00 kdeinit Running...
15797 ?          S      0:00 dcopserver –nosid

and many more...
```

添加 x 选项（注意这里没有前置的连字符）将告知 ps 命令显示所有的进程，而不需要关注它们是由哪个终端（如果有其他的情况）所控制的。TTY 列中出现的“？”表示没有控制终端，使用这个选项可以查看所有进程的列表信息。

由于系统中运行着大量的进程，所以 ps 命令将会输出一个长列表。把 ps 命令的输出作为 less 命令输入的方法通常很管用，它可以更方便地查看显示结果。有些选项组合也会产生很长的输出行，因此最大化终端仿真窗口也是一个好主意。

输出结果中添加了一个命名为 STAT 的新列。STAT 是 state 的缩写，显示的是进程的当前状态，如表 10-1 所示。

表 10-1　进程状态

状态	含义
R	运行状态。进程正在运行或者准备运行
S	睡眠状态。进程不在运行，而是在等待某事件发生，如键盘输入或者收到网络报文
D	不可中断的睡眠状态。进程在等待 I/O 操作，如硬盘驱动
T	暂停状态。进程被指示暂停（后续还可继续运行）
Z	无效或者“僵尸”进程。子进程被终止，但是还没有被其父进程彻底释放掉
<	高优先级进程。进程可以被赋予更多的重要性，分配更多的 CPU 时间。进程的这一特性称为优先级（niceness）。高优先级的进程被说成较不友好，是因为它将消耗更多的 CPU 时间，这样留给其他进程的 CPU 时间就会变少
N	低优先级进程。低优先级进程（友好进程，a nice process）只有在其他更高优先级的进程使用完处理器后才能够获得使用处理器的时间

- fg：设置在前台中运行作业。
- kill：发送信号给某个进程。
- killall：杀死指定名字的进程。
- shutdown：关机或者重启系统。

10.1 进程如何工作

系统启动时，内核先把它的一些程序初始化为进程，然后运行一个称为 init 的程序。init 程序将依次运行一系列称为脚本初始化（init script）的 shell 脚本（放在/etc 目录下），这些脚本将会启动所有的系统服务。其中的很多服务都是通过守护程序（daemon program）来实现的。而后台程序只是呆在后台做它们自己的事情，并且没有用户界面。因此，即使没有用户登录，系统也在忙于执行一些例行程序。

一个程序的运行可以触发其他程序的运行，在进程系统中这种情况被表述为父进程创建子进程。

内核会保存每个进程的信息以便确保任务有序进行。比如，每个进程将被分配一个称为进程 ID（PID，process ID）的号码。进程 ID 是按递增的顺序来分配的，init 进程的 PID 始终为 1。内核也记录分配给每个进程的内存信息以及用来恢复运行的进程就绪信息。和文件系统类似，进程系统中也存在所有者、用户 ID、有效用户 ID 等。

10.1.1 使用 ps 命令查看进程信息

用来查看进程信息的命令中（有多个），使用最普遍的就是 ps 命令。ps 命令有很多选项，其中最简单的使用格式如下所示。

```
[me@linuxbox ~]$ ps
  PID TTY          TIME CMD
5198 pts/1      00:00:00 bash
10129 pts/1     00:00:00 ps
```

这个例子的输出结果列出了两个进程：进程 5198 和进程 10129，它们分别对应 bash 命令和 ps 命令。我们可以发现，默认情况下，ps 命令输出的信息并不是很多，只是输出和当前终端会话相关的进程信息。为了获得更多的信息，我们需要添加一些选项，但是在介绍这个之前，让我们先看看 ps 命令输出的其

系统总体状态信息，下面显示的是一张按 CPU 活动时间排序的进程情况表。

```
top - 14:59:20 up 6:30, 2 users, load average: 0.07, 0.02, 0.00
Tasks: 109 total, 1 running, 106 sleeping, 0 stopped, 2 zombie
Cpu(s):  0.7%us, 1.0%sy,  0.0%ni, 98.3%id, 0.0%wa, 0.0%hi, 0.0%si
Mem:     319496k total,  314860k used,    4636k free,  19392k buff
Swap:    875500k total,  149128k used,  726372k free, 114676k cach

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 6244 me        39  19 31752 3124 2188 S  6.3  1.0 16:24.42  trackerd
11071 me        20   0  2304 1092  840 R  1.3  0.3  0:00.14  top
 6180 me        20   0  2700 1100  772 S  0.7  0.3  0:03.66  dbus-dae
 6321 me        20   0 20944 7248 6560 S  0.7  2.3  2:51.38  multiloa
 4955 root      20   0  104m 9668 5776 S  0.3  3.0  2:19.39  Xorg
    1 root      20   0  2976  528  476 S  0.0  0.2  0:03.14  init
    2 root      15  -5     0    0    0 S  0.0  0.0  0:00.00  kthreadd
    3 root      RT  -5     0    0    0 S  0.0  0.0  0:00.00  migratio
    4 root      15  -5     0    0    0 S  0.0  0.0  0:00.72  ksoftirq
    5 root      RT  -5     0    0    0 S  0.0  0.0  0:00.04  watchdog
    6 root      15  -5     0    0    0 S  0.0  0.0  0:00.42  events/0
    7 root      15  -5     0    0    0 S  0.0  0.0  0:00.06  khelper
   41 root      15  -5     0    0    0 S  0.0  0.0  0:01.08  kblockd/
   67 root      15  -5     0    0    0 S  0.0  0.0  0:00.00  kseriod
  114 root      20   0     0    0    0 S  0.0  0.0  0:01.62  pdflush
  116 root      15  -5     0    0    0 S  0.0  0.0  0:02.44  kswapd0
```

系统总体状态信息包含很多有用的内容，表 10-3 将逐条解释这些字段，具体如下。

表 10-3　顶部信息中的字段

行	字段	含义
1	top	程序名
	14:59:20	一天中的当前时间
	up 6:30	正常运行时间（uptime）。从机器最后一次启动开始计算的时间总数。在这个例子中，系统已经运行了 6.5 小时。
	2 users	有两个用户已登录
	load average:	负载均值（load average）指的是等待运行的进程数；即共享 CPU 资源的处于可运行状态的进程数。显示的三个值分别对应不同的时间段第一个对应的是前 60 秒的均值，下一个对应的是前 5 分钟的均值，最后一个对应的是前 15 分钟的均值。该值小于 1.0 表示该机器并不忙
2	tasks:	统计进程数及各个进程的状态信息
	0.7%us	0.7%的 CPU 时间被用户进程占用，这里指的是处于内核外的进程
	1.0%sy	1.0%的 CPU 时间被系统进程（内核进程）占用
	0.0%ni	0.0%的 CPU 时间被友好进程（nice）（低优先级进程）占用
	98.3%id	98.3%的 CPU 时间是空闲的
	0.0%wa	0.0%的 CPU 时间用来等待 I/O 操作
4	Mem:	显示物理 RAM（随机存取内存）的使用情况
5	Swap:	显示交换空间（虚拟内存）的使用情况

这些进程状态的后面可以带其他的字符来表示不同的特殊进程特性。你可以查看 ps 命令的帮助页面来获取更多的详细信息。

另一个常用的选项组合是 aux（不带前置连字符），它将输出更多的信息，如下所示。

```
[me@linuxbox ~]$ ps aux
USER      PID %CPU %MEM   VSZ   RSS TTY STAT START   TIME COMMAND
root        1  0.0  0.0  2136   644 ?   Ss   Mar05   0:31 init
root        2  0.0  0.0     0     0 ?   S<   Mar05   0:00 [kt]
root        3  0.0  0.0     0     0 ?   S<   Mar05   0:00 [mi]
root        4  0.0  0.0     0     0 ?   S<   Mar05   0:00 [ks]
root        5  0.0  0.0     0     0 ?   S<   Mar05   0:06 [wa]
root        6  0.0  0.0     0     0 ?   S<   Mar05   0:36 [ev]
root        7  0.0  0.0     0     0 ?   S<   Mar05   0:00 [kh]

and many more...
```

该选项组合将会显示属于每个用户的进程信息，使用这些选项时不带前置连字符将使得命令以“BSD 模式（BSD-style）”运行。ps 命令的 Linux 版本可以模拟多种 UNIX 版本中 ps 程序的运行方式，使用这些选项将显示更多列的信息，具体如表 10-2 所示。

表 10-2　BSD 模式下 ps 命令输出的列标题

标题	含义
USER	用户 ID。表示该进程的所有者
%CPU	CPU 使用百分比
%MEM	内存使用百分比
VSZ	虚拟耗用内存大小
RSS	实际使用的内存大小。进程使用的物理内存（RAM）大小（以 KB 为单位）
START	进程开启的时间。如果数值超过 24 个小时，那么将使用日期来显示

10.1.2　使用 top 命令动态查看进程信息

虽然 ps 命令可以显示有关机器运行情况的很多信息，但是它提供的只是在 ps 命令被执行时刻机器状态的一个快照。要查看机器运行情况的动态视图，我们可以使用 top 命令，如下所示。

```
[me@linuxbox ~]$ top
```

top 程序将按照进程活动的顺序，以列表的形式持续更新显示系统进程的当前信息（默认每 3 秒更新一次）。它主要用于查看系统“最高（top）”进程的运行情况，其名字也来源于此。top 命令显示的内容包含两个部分，顶部显示的是

在终端里按下 Ctrl-C 键将会中断（interrupt）一个程序，它意味着我们委婉地请求程序结束。按下 Ctrl-C 键后，xlogo 窗口将关闭，shell 提示符将返回。

许多（但不是所有）命令行程序都可以使用这种方法来实现中断。

10.2.2 使进程在后台运行

假设我们想要 shell 提示符返回，但又不终止 xlogo 程序，那么可以通过让该程序在后台（background）运行来实现。我们可以把终端想象为有一个前台（foreground，表面上可见的内容，类似 shell 提示符）和一个后台（隐藏在表层下面的内容）。要想在启动程序时让该程序在后台运行，可以在命令后面加上和号字符（&）来实现。

```
[me@linuxbox ~]$ xlogo &
[1] 28236
[me@linuxbox ~]$
```

命令执行后，将出现 xlogo 窗口，而且 shell 提示符也将返回，但是同时也会打印一些有趣的数字信息。这条信息是 shell 的一个称为作业控制（job control）的特性表现。shell 通过这条信息来显示已经启动的作业编号为 1（[1]），其对应的 PID 是 28236。如果执行 ps 命令，可以查看到当前运行的进程。

```
[me@linuxbox ~]$ ps
  PID TTY          TIME CMD
10603 pts/1    00:00:00 bash
28236 pts/1    00:00:00 xlogo
28239 pts/1    00:00:00 ps
```

shell 的作业控制特性也提供了一种方式来查看从该终端启动的所有作业。使用 jobs 命令可以得到如下列表信息。

```
[me@linuxbox ~]$ jobs
[1]+ Running                 xlogo &
```

输出结果显示存在一个编号为 1 的作业在运行，而且对应命令是 xlogo &。

10.2.3 使进程回到前台运行

后台运行的进程不会受到任何键盘输入的影响，包括试图用来中断它的 Ctrl-C 键。要想使得进程返回到前台来运行，可以使用 fg 命令来实现，参见下面的例子。

```
[me@linuxbox ~]$ jobs
[1]+ Running                 xlogo &
```

top 程序可以接受许多键盘指令，其中最常用的有两个：一个是 h，输入后将显示程序的帮助界面；另一个是 q，用来退出 top 命令。

主流的桌面环境都提供了用来显示类似 top 命令的输出信息的图形化应用程序（和 Windows 中任务管理器[Task Manager]的运行方式类似），但是 top 命令优于图形化版本，这是因为 top 命令运行得更快，而且消耗的系统资源要少得多。毕竟，系统监控程序不应该减缓正在被监控的系统的处理速度。

10.2 控制进程

既然我们已经知道了如何查看和监控进程，那么接下来让我们看看如何对进程进行控制。我们将使用一个称为 xlogo 的小程序作为实验对象。xlogo 程序是由 X 窗口系统（X Window System，使得显示器支持图形化界面的底层引擎）提供的一个示例程序，它只简单地显示一个包含 X 标识的可缩放窗口。首先，我们认识一下实验对象。

```
[me@linuxbox ~]$ xlogo
```

输入该命令后，包含该标识的一个小窗口将在屏幕的某个地方出现。有些系统中，xlogo 可能会输出一条告警信息，但是我们可以忽略它，因为它并不会造成什么影响。

注意 如果系统中不包含 xlogo 程序，那么试着使用 gedit 程序或者 kwrite 程序来替代。

我们可以通过改变窗口的大小来验证 xlogo 是否处于运行状态。如果该标识适应新的窗口大小被重新绘制了，则表明该程序正在运行。

注意，为什么这里 shell 提示符没有返回呢？这是因为 shell 正在等待该 xlogo 程序结束，就像以前使用的其他程序一样。如果关闭 xlogo 窗口，那么提示符将返回。

10.2.1 中断进程

让我们观察再次运行 xlogo 命令的时候会发生什么。首先，输入 xlogo 命令，并确保程序在正常运行。接下来，返回到终端窗口，按下 Ctrl-C 键。

```
[me@linuxbox ~]$ xlogo
[me@linuxbox ~]$
```

序或者反过来拒绝终止的程序。这里有一个例子，如下所示。

```
[me@linuxbox ~]$ xlogo &
[1] 28401
[me@linuxbox ~]$ kill 28401
[1]+ Terminated                  xlogo
```

我们首先在后台启动了 xlogo 程序。shell 将打印输出该后台进程的 jobspec 选项信息和 PID 信息。接着，我们使用了 kill 命令，并且指定想要终止进程的 PID。我们也可以使用 jobspec 选项（例如，%1）代替 PID 信息来指定该进程。

这些看起来都非常简单，但是事实上，它们包含着更多的内容。kill 命令准确地说并不是“杀死”进程，而是给进程发送信号（signal）。信号是操作系统和程序间通信的多种方式之一，在使用 Ctrl-C 键和 Ctrl-Z 键时已经见识过信号的作用。当终端接收到其中的一个输入时，它将发送信号到前台进程。在按下 Ctrl-C 键的情况下，它将发送一个称为 INT（中断，Interrupt）的信号；在按下 Ctrl-Z 的情况下，它将发送一个称为 TSTP（终端暂停，Terminal Stop）的信号。反过来，程序“侦听”信号，而且在接收到信号的时候按照它们的指示进行操作。程序可以侦听信号并且可以按照信号指示操作的这一特性，使得程序在接收到终止信号的时候可以保存当前正在进行的工作。

10.3.1 使用 kill 命令发送信号到进程

kill 命令最常用的语法格式如下。

```
kill [-signal] PID...
```

如果命令行中没有指定信号，那么默认发送 TERM（终止，Terminate）信号。kill 命令最常用来发送的信号如表 10-4 所示。

表 10-4 常用信号

信号编号	信号名	含义
1	HUP	挂起信号。这是美好的过去留下的痕迹，当时通过电话线和调制解调器来把终端和远端计算机连接在一起。该信号用来指示程序控制终端已被“挂起”。该信号的效果通过关闭终端会话的方式来表现。运行在终端上的前台程序收到该信号后将终止。该信号也被很多后台程序用来进行重新初始化。这就意味着，当一个后台进程接收到该信号时，它将重启并且重新读取它的配置文件。Apache Web 服务器就是后台进程使用 HUP 信号重新初始化的一个例子
2	INT	中断信号。执行效果和在终端按下 Ctrl-C 键的效果一样。通常用来终止一个程序

```
[me@linuxbox ~]$ fg %1
xlogo
```

我们可以通过在 fg 命令后面加上百分比符号和作业编号（称为 jobspec 选项）来实现这个功能。如果后台只有一个任务，那么可以不带 jobspec 选项。这个时候按下 Ctrl-C 键就可以终止 xlogo 命令。

10.2.4 停止（暂停）进程

如果我们只是想要暂停进程，而不是终止进程，那么通常需要我们将前台运行的进程移到后台去运行。我们为了暂停前台进程需要按下 Ctrl-Z 键。让我们试试如下操作，在命令提示符后输入 xlogo，按下 Enter 键后再按下 Ctrl-Z 键。

```
[me@linuxbox ~]$ xlogo
[1]+ Stopped                 xlogo
[me@linuxbox ~]$
```

在暂停 xlogo 命令后，我们可以通过试图改变 xlogo 窗口的大小来确认该程序是否真正被暂停了。可以发现，该进程看起来好像死了。这个时候，我们可以使用 fg 命令让进程在前台恢复运行，也可以使用 bg 命令让进程移到后台运行：

```
[me@linuxbox ~]$ bg %1
[1]+ xlogo &
[me@linuxbox ~]$
```

在使用 fg 命令的时候，如果只存在一个作业，那么可以不带 jobspec 选项。

如果用命令方式启动了一个图形化程序，但是忘记了在命令尾部加上“&”符号来让程序在后台运行，那么在这种情况下，把进程从前台移到后台去运行的方法将非常方便。

为什么会想要通过命令行的方式来启动一个图形化程序呢？原因有两个。首先，想要运行的程序可能并不在窗口管理器的菜单中（比如 xlogo 程序）。

其次，从命令行启动程序可以看到用图形化方式启动程序所看不到的错误信息。有时候从图形菜单中启动程序，程序会启动失败。但改用命令行方式启动的话，就可以得到错误提示信息，找到问题所在。另外，一些图形化程序也包含很多有意思的和有用的命令行选项。

10.3 信号

kill 命令通常用来“杀死”（终止）进程，它可以用来终止运行不正常的程

表 10-5 其他常用信号

信号编号	信号名	含义
3	QUIT	退出信号
11	SEGV	段错误信号。如果程序非法使用了内存空间，即程序试图在没有写权限的空间执行写操作，那么系统将发送该信号
20	TSTP	终端暂停信号。在按下 Ctrl-Z 键时终端将发出该信号。与 STOP 信号不同的是，TSTP 信号由程序接收，但是程序可以选择忽略该信号
28	WINCH	窗口改变信号。当窗口改变大小时，系统将发送该信号。类似 top 和 less 的一些程序将会对该信号作出响应，重新绘制视图来适应新的窗口大小

如果想要查看更多的信号，使用如下命令将显示完整的信号列表。

```
[me@linuxbox ~]$ kill -l
```

10.3.2 使用 killall 命令发送信号给多个进程

通过使用 killall 命令，我们可以给指定程序或者指定用户名的多个进程发送信号。一般语法格式如下。

```
killall [-u user] [-signal] name...
```

要证明这一点，我们可以先启动两个 xlogo 程序实例，然后终止它们。

```
[me@linuxbox ~]$ xlogo &
[1] 18801
[me@linuxbox ~]$ xlogo &
[2] 18802
[me@linuxbox ~]$ killall xlogo
[1]- Terminated                    xlogo
[2]+ Terminated                    xlogo
```

记住，和 kill 命令一样，你必须具有超级用户权限，才能够使用 killall 命令给不属于自己的进程发送信号。

10.4 更多与进程相关的命令

由于进程监控是一项重要的系统管理任务，所以存在很多命令用来为它服务。表 10-6 列出了其中一些命令。

续表

信号编号	信号名	含义
9	KILL	杀死信号。该信号比较特殊。鉴于程序可以选择不同的方式来处理发送过来的信号，包括忽略所有的这些信号，KILL 信号将不会真正意义上地被发送到目标程序。而是内核宁愿立即终止了该进程。当进程以这种方式被终止时，它将没有机会对它自己进行“清理”或者对当前工作进行保存。考虑到这个原因，KILL 信号只能当作其他的终端信号都执行失败的情况下的最后选择
15	TERM	终止信号。这是 kill 命令默认发送的信号类型。如果程序仍然有足够的“活力”（alive enough）来接收信号，那么它将被终止
18	CONT	继续运行信号。恢复之前接受了 STOP 信号的进程
19	STOP	暂停信号。该信号将使进程暂停，而不是终止。和 KILL 信号类似，该信号不会被发送给目标进程，因此它不能被忽略

按照下面的方式使用 kill 命令。

```
[me@linuxbox ~]$ xlogo &
[1] 13546
[me@linuxbox ~]$ kill -1 13546
[1]+ Hangup                    xlogo
```

在这个例子中，我们首先在后台启动了 xlogo 程序，接着使用 kill 命令给它发送 HUP 信号。xlogo 程序将终止，shell 的输出信息表明这个后台进程已经接收了一个挂起信号。你也许需要多敲几次 Enter 键才能看到这条输出信息。注意，你可以通过信号编号或者信号名来指定信号，其中包含带有 SIG 前缀的信号名。

```
[me@linuxbox ~]$ xlogo &
[1] 13601
[me@linuxbox ~]$ kill -INT 13601
[1]+ Interrupt                 xlogo
[me@linuxbox ~]$ xlogo &
[1] 13608
[me@linuxbox ~]$ kill -SIGINT 13608
[1]+ Interrupt                 xlogo
```

尝试使用其他的信号重复执行上面的例子。记住，你也可以使用 jobspec 选项来代替 PID 信息。

和文件一样，进程也有所有者，只有进程的所有者（或者超级用户）才能使用 kill 命令来给它发送信号。

除了表 10-4 中列出的通常用于 kill 命令的信号之外，还存在其他一些经常被系统使用的信号。表 10-5 列出的是其他的一些常用信号。

第二部分

配置与环境

表 10-6　其他与进程相关的命令

命令	描述
pstree	以树状的模式输出进程列表，该模式显示了进程间的父/子关系
vmstat	输出系统资源使用情况的快照，包括内存，交换空间和磁盘 I/O。如果想要持续查看输出，可以在命令后面加上一个间隔时间（以秒为单位），命令将按照间隔时间来动态更新显示的内容（比如，vmstat 5）。按下 Ctrl-C 键可以终止输出
xload	用来绘制显示系统时间负载情况图形的一种图形化界面程序
tload	类似于 xload 程序，但是图形是在终端上绘制。按下 Ctrl-C 键终止输出

第11章 环　境

前面讲到，在 shell 会话调用环境（environment）期间，shell 会存储大量的信息。程序使用存储在环境中的数据来确定我们的配置。尽管大多数系统程序使用配置文件（configuration file）来存储程序设置，但是也有一些程序会查找环境中存储的变量来调整自己的行为。知道这一点之后，用户就可以使用环境来自定义 shell。

本章会讲解下述命令。

- printenv：打印部分或全部的环境信息。
- set：设置 shell 选项。
- export：将环境导出到随后要运行的程序中。
- alias：为命令创建一个别名。

11.1 环境中存储的是什么

尽管 shell 在环境中存储了两种基本类型的数据，但是在 bash 中，这两种类

```
[me@linuxbox ~]$ set | less
```

与 printenv 命令不同的是，set 命令的输出结果是按照字母顺序排列的。

如需要查看单个变量的值，我们也可以使用 echo 命令，如下所示。

```
[me@linuxbox ~]$ echo $HOME
/home/me
```

set 命令和 printenv 命令都不能显示的一个环境元素是别名。要查看别名，需使用不带任何参数的 alias 命令。

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --showtilde'
```

11.1.2 一些有趣的变量

环境中包含了相当多的变量，尽管你所使用的环境与这里的不相同，也会在你的环境中看到表 11-1 中所示的变量。

表 11-1 环境变量

变量	说明
DISPLAY	运行图形界面环境时界面的名称。通常为:*0*，表示由 X 服务器生成的第一个界面
EDITOR	用于文本编辑的程序名称
SHELL	本机 shell 名称
HOME	本机主目录的路径名
LANG	定义了本机语言的字符集和排序规则
OLD_PWD	先前的工作目录
PAGER	用于分页输出的程序名称。通常设置为/usr/bin/less
PATH	以冒号分割的一个目录列表，当用户输入一个可执行程序的名称时，会查找该目录列表
PS1	提示符字符串 1。定义了本机 shell 系统提示符的内容。在后面我们会看到，可以灵活地自定义该变量
PWD	当前工作目录
TERM	终端类型的名称。类 UNIX 系统支持很多种终端协议；此变量设定了本机终端模拟器使用的协议
TZ	用于指定本机所处的时区。大多数类 UNIX 系统以协调世界时（UTC）来维护计算机的内部时钟，而显示的本地时间是根据本变量确定的时差计算出来的
USER	用户名

如果某些变量无法在该表中找到也不要紧，因为这些变量会因发行版本的不同而有差异。

11.2 环境是如何建立的

用户登录系统后，bash 程序就会启动并读取一系列称为启动文件的配置脚本，这些脚本定义了所有用户共享的默认环境。接下来，bash 会读取更多存储在主目录下的用于定义个人环境的启动文件。这些步骤执行的确切顺序是由启动的 shell 会话类型决定的。

11.2.1 login 和 non-login shell

shell 会话存在两种类型，分别为 login shell 会话和 non-login shell 会话。

login shell 会话会提示用户输入用户名和密码，如虚拟控制台会话。而我们在 GUI 中启动的终端会话就是一个典型的 non-login shell 会话。

login shell 会读取一个活多个启动文件，如表 11-2 所示。

表 11-2 login shell 的启动文件

文件	说明
/etc/profile	适用于所有用户的全局配置脚本
~/.bash_profile	用户的个人启动文件。可扩展或重写全局配置脚本中的设置
~/.bash_login	若~/.bash_profile 缺失，则 bash 尝试读取此脚本
~/.profile	若~/.bash_profile 与~/.bash_login 均缺失，则 bash 尝试读取此文件。在基于 Debian 的 Linux 版本中（比如 Ubuntu），这是默认值

表 11-3 所示为 non-login shell 读取的启动文件。

表 11-3 non-login shell 的启动文件

文件	内容
/etc/bash.bashrc	适用于所有用户的全局配置脚本
~/.bashrc	用户的个人启动文件。可扩展或重写全局配置脚本中的设置

在读取以上启动文件之外，non-login shell 还会继承父类进程的环境，父类进程通常是一个 login shell。

用户可查看本机系统有哪些启动文件，需要注意的是这些文件大多数以“.”

开头（意味着这些文件是被隐藏的），所以用户在使用 ls 命令时，需要伴随使用 -a 选项。

在普通用户看来，~/.bashrc 可能是最重要的启动文件，因为系统几乎总是要读取它。non-login shell 会默认读取~/.bashrc，而大多数 login shell 的启动文件也能以读取~/.bashrc 文件的方式来编写。

11.2.2 启动文件中有什么

一个典型的.bash_profile（来自于 CentOS-4 系统）内容如下所示。

```
# .bash_profile
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi
# User specific environment and startup programs

PATH=$PATH:$HOME/bin
export PATH
```

文件中以“#”开始的行是注释行，而 shell 是不会读取注释行的，注释是为提高用户可读性而存在的。一件有趣的事发生在第 4 行，如下所示。

```
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi
```

这段代码被称作 if 复合命令，会在本书的第四部分进行讲解，现在可以将这段代码理解为如下所示的内容。

```
If the file "~/.bashrc" exists, then
        read the "~/.bashrc" file.
```

可以看到这一段代码阐述了 login shell 读取.bashrc 文件的机制。以上启动文件中另一个很重要的元素是 PATH 变量。

在命令行输入一条命令后，你曾经疑惑过 shell 是怎样找到这些命令的吗？当用户输入命令 ls，shell 不会搜索整个系统来寻找/bin/ls（ls 命令的完整路径名），而是会搜索 PATH 变量中存储的目录列表。

PATH 变量通常是由启动文件/etc/profile 中的一段代码设定（并不总是如此，这取决于系统的发行版本）。

```
PATH=$PATH:$HOME/bin
```

这段代码将$HOME/bin 添加到了 PATH 值的尾部。这是一个参数扩展的

实例，本书第 7 章介绍过相关的内容。以下代码可以帮助用户理解参数扩展的机理。

```
[me@linuxbox ~]$ foo="This is some"
[me@linuxbox ~]$ echo $foo
This is some
[me@linuxbox ~]$ foo=$foo" text."
[me@linuxbox ~]$ echo $foo
This is some text.
```

使用参数扩展，用户可以将更多的内容添加到变量值的尾部。

在把字符串$HOME/bin 添加到 PATH 值的尾部之后，当系统需要检索用户输入的命令时，$HOME/bin 这个路径就会处于被搜索的路径列表中。这就意味着当我们想在主目录下创建名为 *bin* 的目录，并在此目录中存放自己的私有程序时，shell 已经为我们准备好了，我们要做的就是将创建的目录称之为 bin。

注意 很多 Linux 发行版本在默认情况下提供了该 PATH 设置。一些基于 Debian 的发行版本，如 Ubuntu，会在登录时检查~/bin 目录是否存在，若存在，就会自动将其添加到 PATH 变量中。

最后一行是如下代码：

```
export PATH
```

该 export 命令告诉 shell，将 shell 的子进程使用 PATH 变量的内容。

11.3 修改环境

现在用户已经知道了系统启动文件的位置和内容，就可以修改启动文件，来自定义我们的环境。

11.3.1 用户应当修改哪些文件

一般来说，在 PATH 中添加目录，或者定义额外的环境变量，需要将这些更改放入到.bash_profile 文件中（或者是其他的等效文件，这取决于系统的发行版本，比如 Ubuntu 系统使用的是.profile 文件），其他的改变则应录入.bashrc 文件中。除非是系统管理员需要修改用户公用的默认设置，普通用户只需对主目录下的文件作出修改即可。当然用户也可以修改其他目录下的文件，比如/etc 下的 profile 文件，而且很多情况会需要用户这样做，但是现在我们先保险一点操作。

11.3.2　文本编辑器

为了编辑（比如修改）shell 的启动文件，以及系统中的其他大多数配置文件，我们会用到一个称为文本编辑器的程序。文本编辑器类似于字处理器，它允许用户通过移动移动光标的方式来编辑屏幕中的文字。与字处理器不同的是，文本编辑器只支持纯文本，而且通常包含为编写程序而设计的特性。文本编辑器是软件开发人员编写代码的主要工具，系统管理员也可以使用文本编辑器来管理系统的配置文件。

Linux 系统可使用的文本编辑器有很多种，你的系统中可能装有不止一种的文本编辑器。为什么会有这么多种编辑器？主要是因为程序员热衷于编写文本编辑器，既然程序员在工作中会广泛地用到编辑器，他们希望文本编辑器能符合自己的工作方式。

文本编辑器可大概分为两类：图形界面的和基于文本的。GNOME（GNU 网络对象模型环境）和 KDE（K 桌面环境）都配备有一些流行的图形界面编辑器。GNOME 配备的编辑器叫做 gedit，在 GNOME 菜单中 gedit 通常被称为 Text Editor。KDE 则配备了三种编辑器，分别是 kedit、kwrite 和 kate（复杂程度递增）。

有很多种基于文本的编辑器，常见编辑器中较受用户欢迎的是 nano、vi 和 emacs。nano 是一种简单易用的编辑器，最初是为了替代 pico（由 PINE 电子邮件套件提供）而出现的。vi 是类 Unix 系统的传统文本编辑器（在大多数 Linux 系统中已被 vim——Vi Improved 的缩写——所取代），本书第 12 章的讲解主题就是 vi。而 emacs 编辑器最初由 Richard Stallman 编写，这是一个庞大的、万能的、可做任何事情的编辑环境。尽管 emacs 仍然可用，但是大多数 Linux 系统很少默认安装 emacs。

11.3.3　使用文本编辑器

所有的文本编辑器都可以通过在命令行输入编辑器名称和需编辑的文件名称的方式启动。如果输入的文件不存在，编辑器会认为用户想要创建一个新的文件。下面是一个使用 gedit 的范例。

```
[me@linuxbox ~]$ gedit some_file
```

如果 some_file 文件存在，这条命令将启动 gedit 编辑器，并载入 some_file。

因为所有的图形界面编辑器都非常易于理解，所以这里就不做赘述。接下

来，我们将通过.bashrc 文件的编辑过程来讲解 nano，nano 是第一个基于文本的文本编辑器。但在此之前，需要先采取一些安全措施。在修改一些重要的配置文件时，先对配置文件进行备份再进行编辑是一个很好的习惯。当用户把文件修改得一团糟的时候，备份就很有用处了。我们可以使用以下代码备份.bashrc：

```
[me@linuxbox ~]$ cp .bashrc .bashrc.bak
```

为备份文件取什么名字并不重要，只要备份文件的名称易于理解即可。扩展名.bak、.sav、.old 和.orig 是常用的标示备份文件的方法。需要说明的是，cp 命令会默默地覆盖现有的文件。

备份文件完成之后，就可以启动文件编辑器了。

```
[me@linuxbox ~]$ nano .bashrc
```

nano 启动后，屏幕显示内容如下所示。

```
  GNU nano 2.0.3          File: .bashrc

# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

# User specific aliases and functions

                          [ Read 8 lines ]
^G Get Help^O WriteOut^R Read Fil^Y Prev Pag^K Cut Text^C Cur Pos
^X Exit    ^J Justify ^W Where Is^V Next Pag^U UnCut Te^T To Spell
```

注意 *若系统没有安装 nano，也可以使用图形界面编辑器来进行操作。*

屏幕显示内容分为三部分：顶端的标题（header）、中间的可编辑文本和底部的命令菜单。由于 nano 是替代电子邮件文本编辑器出现的，所以其编辑功能非常有限。

对于每一种文本编辑器，你都应该首先学习它的退出命令。就 nano 来说，可按 Ctrl-X 退出程序，在页面底部的命令菜单中有相关的介绍。“^X”代表了 Ctrl-X，这是控制字符的常见表示法，很多程序中都使用它。

我们需要了解的第二个命令就是如何保存我们的工作。就 nano 来说，按 Ctrl-O 完成保存。掌握这些知识之后，我们就可以进行文本编辑操作了。请使用向下箭头键或者向下翻页键使光标移动到文件的末尾，然后添加以下代码到.bashrc 文件中。

```
umask 0002
export HISTCONTROL=ignoredups
export HISTSIZE=1000
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
```

注意 用户系统的.bashrc 文件可能已经写入了这些代码的一部分，但是不用担心，重复的代码不会造成什么危害。

表 11-4 列出了以上代码的含义。

表 11-4 .bashrc 文件增加的代码

代码行	含义
Umask 0002	设置 umask 值以解决第 9 章中讨论过的共享目录的问题
Export HiSTCONTROL=ignoredups	使 shell 的历史记录功能忽略与上一条录入的命令重复的命令
Export HISTSIZE=1000	使命令历史记录规模从默认的 500 行增加到 1000 行
alias l.='ls -d .* --color=auto'	创建新的命令：l.，功能是显示所有以 . 开头的目录条目
alias ll='ls -l -color=auto'	创建新的命令：ll，功能是以长格式来展示目录列表

可以看到，很多新增加的代码并不易于理解，所以就需要在.bashrc 文件中添加一些注释来帮助用户理解代码的含义。添加注释后的代码如下所示。

```
# Change umask to make directory sharing easier
umask 0002

# Ignore duplicates in command history and increase
# history size to 1000 lines
export HISTCONTROL=ignoredups
export HISTSIZE=1000

# Add some helpful aliases
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
```

这样一来就易懂多了。最后我们按 Ctrl-O 保存文档，按 Ctrl-X 退出 nano，这样对.bashrc 文件的修改就完成了。

11.3.4 激活我们的修改

因为只有在启动 shell 会话时才会读取.bashrc，所以对.bashrc 做出的修改只有在关闭 shell 终端会话并重启的时候才会生效。当然也可以使用以下命令强制

命令 bash 重新读取.bashrc 文件。

```
[me@linuxbox ~]$ source .bashrc
```

重新读取.bashrc 之后，文件中做出的修改就会生效。我们来试一下其中的一个新的别名。

```
[me@linuxbox ~]$ ll
```

为什么注释很重要

不管何时修改配置文件，随手添加一些注释来记录做出的改变是一个好的习惯。我们可以记住近期做出的修改，但是 6 个月之前的修改呢？所以，顺手写上一些注释吧，这对用户本身有益。同时，在写注释的时候，附加一个修改记录是一个不错的主意。

在 shell 脚本和 bash 启动文件中，注释是以“#”开头的。其他的配置文件可能会使用其他的符号。大多数配置文件中都有注释，这些注释能起到很好的向导作用。

在配置文件中经常会看到一些代码被注释掉，以防止它们被相关程序读取。这是为了给读者示范可能的配置选项或者正确的配置方法。比如，Ubuntu 8.04 的.bashrc 文件包含以下内容。

```
# some more ls aliases
#alias ll='ls -l'
#alias la='ls -A'
#alias l='ls -CF'
```

最后三行被注释掉的代码定义的别名是有效的。如果将这三行代码去注释化，也就是去掉开头的“#”符号，那么这些别名将被激活。反之，如果在一行代码前添加“#”符号，就可以在保留原本信息的基础上使这行配置代码失效。

11.4 本章结尾语

本章节讲解了一项重要的基本技能——使用文本编辑器编辑配置文件。随着学习的继续，当我们查看某一命令的 man 文档时，记录下该命令支持的环境变量，可能会有有趣的发现。接下来的章节会讲到 shell 函数，shell 函数是一种强大的功能，用户可将其添加到 bash 启动文件中，以此来添加你的自定义命令。

第 12 章 VI 简介

有一个古老的笑话，讲的是一个观光客想去纽约著名的古典音乐厅，于是找到一个路人问路的故事。

观光客："不好意思，请问怎样才能到达卡耐基音乐厅？"

路人："你要练习，练习，再练习！"

就像一个人不可能一夕之间成为技艺高超的钢琴家，Linux 命令也不是花一个下午就能熟练掌握的，这需要很长时间的练习。本章节将介绍 UNIX 传统核心软件之一——文本编辑器 vi（发音是"vee eye"）。vi 用户界面的不友好是非常出名的，但是看一位 vi 专家在键盘前坐下并开始"演奏"，将是莫大的艺术享受。本章节并不能使读者成为 vi 专家，但是在学习之后，读者至少能够做到在 vi 中演奏"Chopsticks"。

12.1 为什么要学习 vi

现在这个时代存在着很多图形界面编辑器和易用的基于文本的编辑器，例

如 nano，那为什么还要学习 vi？这有三条充分的理由。

- vi 总是可用的。如果用户面前的系统没有图形界面，例如是远程服务器或者是本地系统的 X 配置不可用，那么 vi 就会成为救命的稻草。尽管 nano 已经得到了越来越广泛的应用，但是，迄今为止它还不是通用的。而 POSIX（一种 UNIX 系统的程序兼容标准）则要求系统必须配备有 vi。
- vi 是轻量级的软件，运行速度快。对很多任务来说，启动 vi 比在菜单中找到一个图形界面编辑器并等待几兆大小的编辑器载入要容易得多。另外，vi 的设计还非常利于打字。在接下来的讲解中读者可以了解到，vi 高手在编辑过程中甚至不需要把手指从键盘上移开。
- 用户不想被其他 Linux 和 UNIX 用户蔑视。

好吧，也许只有两条正当的理由。

12.2 VI 背景

1976 年，加州大学伯克利分校的学生，之后又成为 Sun 公司创始人之一的 Bill Joy 写出了 vi 的第一个版本。vi 出自单词“visual”，含义是能够在视频终端上用移动光标来进行编辑。在图形界面编辑器出现之前是行编辑器的天下，用户每次只能在一行文本上进行编辑。使用行编辑器的时候，用户需要告知编辑器是在哪一行进行什么样的操作，比如添加或者删除。而视频终端（而非基于打印机的终端，比如电报）的来临使得全屏幕编辑成为可能。由于 vi 融合了强大的行编辑器 ex，vi 用户也可以同时使用行编辑的命令。

大多数 Linux 发行版配备的并不是真正的 vi，而是 Bram Moolenaar 编写的 vi 加强版——vim（Vi Improved 的缩写）。vim 是传统 UNIX 系统中 vi 的实质性改良版。通常，vin 的硬连接（或别名）指向 Linux 系统的 vi 名称。接下来的讨论就是建立在用户使用名为 vi 的 vim 程序这样一个假设上的。

12.3 启动和退出 vi

输入以下命令启动 vi：

```
[me@linuxbox ~]$ vi
```

屏幕显示如下：

```
~
~
~                   VIM - Vi Improved
~
~                    version 7.1.138
~               by Bram Moolenaar et al.
~      Vim is open source and freely distributable
~
~               Sponsor Vim development!
~     type :help sponsor<Enter>    for information
~
~     type :q<Enter>               to exit
~     type :help<Enter> or <F1>    for on-line help
~     type :help version7<Enter>   for version info
~
~            Running in Vi compatible mode
~     type :set nocp<Enter>        for Vim defaults
~     type :help cp-default<Enter> for info on this
~
~
~
```

像操作 nano 一样，现在应该先学习如何退出 vi。输入以下命令退出 vi（需要注意的是，冒号是命令的一部分）。

```
:q
```

此时 shell 会返回初始的操作窗口。如果因为一些原因，vi 不能够退出（通常是因为没有保存修改过的文件），可以通过在命令后添加感叹号的方式强制退出 vi。

```
:q!
```

注意 如果用户不能确定 vi 所处的状态，可以按 Esc 键两次返回初始状态。

12.4 编辑模式

再次启动 vi，并向其传递一个不存在的文件名，就可以通过 vi 创建新文件。

```
[me@linuxbox ~]$ rm -f foo.txt
[me@linuxbox ~]$ vi foo.txt
```

正常情况下，屏幕显示如下所示。

```
~
~
~
```

```
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"foo.txt" [New File]
```

每行开头的波浪线代表此行没有任何内容，也就是说现在 foo 是空白的文件。先不要输入任何内容！

讲解过如何退出 vi 之后，接下来需要了解的就是 vi 是一个模态编辑器。vi 启动后进入的是命令模式。在命令模式中，几乎键盘上的每一个按键都代表一条命令，所以在这时对 vi 进行普通输入的话，vi 基本上就要崩溃了，并且会把文件弄得一团糟。

12.4.1 进入插入模式

如果用户需要向文件中添加一些内容，那么首先要做的就是按 I 键（或 i）进入插入模式。若此时 vim 是在增强模式下正常地运行，那么在屏幕底部会出现以下内容（若 vim 以兼容模式运行，则不会出现）：

```
-- INSERT --
```

现在用户可以进行输入操作了，例如：

```
The quick brown fox jumped over the lazy dog.
```

最后按 Esc 键退出插入模式并返回命令模式。

12.4.2 保存工作

要保存用户修改过的文件，在命令模式下输入一条 ex 命令，也就是按“:”键。这样之后，一个冒号会出现在屏幕的底部：

```
:
```

要将文件写入硬盘，在冒号之后输入 w，如下所示：

```
:w
```

文件写入硬盘驱动器之后，用户会在屏幕底部得到一条确认信息。

```
"foo.txt" [New] 1L, 46C written
```

注意 如果用户阅读 vim 的说明文档，会困惑地发现命令模式被称为普通模式，而使用 ex 命令则被称为命令模式。这方面需多加留意。

兼容模式

在前面示例的 vi 启动屏幕（取自 Ubuntu 8.04 版本）中，用户可以看到这样的内容：Running in Vi compatible mode。这意味着 vim 将以近似于 vi 的常规模式运行，而不是加强版的 vim 行为。为达到本章的目的，用户需要使用加强模式下的 vim。以下两种方式都可以达到此目的。

- 运行 vim 而不是 vi（如果此法可行，可以考虑在.bashrc 文件中添加别名 vi='vim'）。
- 使用以下命令在 vim 配置文件中添加一行内容。

```
echo "set nocp" >> ~/.vimrc
```

Linux 发行版本不同，其 vim 包也就不同。一些版本在默认情况下只是安装了 vim 的最小版本，只支持有限的 vim 特性。在接下来的讲解中，可能会用到 vim 特性缺少的情况。如果是这样的话，你可安装一个完全版的 vim。

12.5 移动光标

在命令模式下，vi 提供了很多移动光标命令，其中有一些命令是与 less 命令共用的。表 12-1 列出了命令的一部分。

表 12-1 光标移动功能键

键	光标动作
L 或右方向键	右移一位
H 或左方向键	左移一位
J 或下方向键	下移一行
K 或上方向键	上移一行
数字 0	至本行开头

续表

键	光标动作
Shift-6(^)	至本行第一个非空字符
Shift-4($)	至本行的末尾
W	至下一单词或标点的开头
Shift-W(W)	至下一单词的开头，忽略标点
B	至上一单词或标点的开头
Shift-B(B)	至上一单词的开头，忽略标点
Ctrl-F 或 Page Down	下翻一页
Ctrl-B 或 Page UP	上翻一页
number-Shift-G	至第 number 行（如 1G 会将光标移到文件的第一行）
Shift-G(G)	至文件的最后一行

为什么使用 H、J、K 和 L 键来移动光标呢？这是因为在 vi 最初出现的阶段，并不是所有的视频终端都有方向键，这样的设计使得 vi 高手可以手不离键盘地移动光标。

像表 12-1 的 G 命令一样，许多 vi 的命令的前面都可以缀上数字。前缀数字可以控制命令执行的次数，比如 5j 可以使得光标下移 5 行。

12.6 基本编辑

插入、删除、剪切、复制等构成了基本的文本编辑操作，vi 也以其特殊的方式支持这些操作。同时 vi 还支持有限形式的撤销操作，在命令模式下按 U 键就可以撤销用户最后一步操作。这项功能在学习一些编辑命令的时候会很有帮助。

12.6.1 添加文本

有几种方式都可以进入 vi 的插入模式。现在假设已经使用 i 命令进入插入模式。

先回顾下 foo.txt 内容。

```
The quick brown fox jumped over the lazy dog.
```

因为光标不能跳出行末，所以单纯使用 i 命令并不能完成在文本末尾添加内容的任务。为此 vi 提供了在行末添加文本的 a 命令。当用户将光标移动到行的

末尾并使用 a 命令时，光标就会越过文本的末尾，同时 vi 进入插入模式。这样用户就可以在行末添加文本了。

```
The quick brown fox jumped over the lazy dog. It was cool.
```

输入结束后不要忘记按 Esc 键退出插入模式。

因为用户经常用到在行末添加文本的功能，所以 vi 提供了使光标移动到行末并进入插入模式的快捷方式——A 命令。现在我们就来试一下。

首先，使用 0 命令将光标移动到行的开头。接下来使用 A 命令将以下内容写入文件中。

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
Line 5
```

按 Esc 键退出插入模式。

可以看到，A 命令使 vi 进入插入模式并自动将光标移动到行尾，非常好用。

12.6.2 插入一行

插入文本的另一种方式是在文本中重开一行，即在两行现存的文字中间插入空白行并进入插入模式。表 12-2 列出了插入一行的两种方式。

表 12-2 插入一行功能键

命令	开行
o	当前行的上方
O	当前行的下方

下面这个例子示范了这两种命令的作用。先将光标置于 Line 3，再输入 o，结果如下所示。

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3

Line 4
Line 5
```

我们可以看到在第三行的下方 vi 插入了一行，并进入了插入模式。按 Esc 键退出插入模式，按 u 键取消上述操作。

继续输入命令 o，就会在第三行的上方插入了一行。

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2

Line 3
Line 4
Line 5
```

再次按 Esc 键退出插入模式并按 u 键取消操作。

12.6.3 删除文本

就像用户期望的一样，vi 提供了很多种删除文本的方式，每一种都需要进行一次至两次的按键操作。首先，X 键会删除光标处的字符。x 命令可加以数字前缀来明确删除的字符数目。D 键则使用得更加普遍。像 x 命令一样，d 命令也可加练数字前缀来明确删除的次数。另外，d 命令总是加以控制删除范围的光标移动命令作为后缀。表 12-3 给出了一些范例。

现在我们进行命令练习。我们将光标移至文件首行单词 It 的首字母，使用 x 命令直到完全删除本句。然后按 u 键直到所有的删除操作都被取消为止。

注意 实际上，vi 只能取消一次操作，vim 可取消多次操作。

表 12-3 文本删除命令

命令	删除内容
x	当前字符
3x	当前字符和之后 2 个字符
dd	当前行
5dd	当前行和之后 4 行
dW	当前字符到下一单词的起始
d$	当前字符到当前行的末尾
d0	当前字符到当前行的起始
d^	当前字符到当前行下一个非空字符
dG	当前行到文件末尾
d20G	当前行到文件第 20 行

现在让我们练习使用 d 命令。我们再次将光标移动到单词 It，使用 dW 命令来删除整个单词。

```
The quick brown fox jumped over the lazy dog. was cool.
Line 2
Line 3
Line 4
Line 5
```

使用的 d$删除光标至本行末尾的字符。

```
The quick brown fox jumped over the lazy dog.
Line 2
Line 3
Line 4
Line 5
```

使用 dG 删除当前行到文件末尾的内容。

```
~
~
~
~
~
```

使用 u 命令三次来取消以上操作。

12.6.4　剪切、复制和粘贴文本

命令 d 不只是删除文本，而是在“剪切”文本。用户每次使用 d 命令之后，都会复制删除的内容进缓存（类似剪贴板），然后用户就可以使用 p 命令将缓存中的内容粘贴到光标之后或使用 P 命令将内容粘贴到光标之前。

就像命令 d 剪切文本的形式一样，命令 y 会“复制”文本。表 12-4 列举了一些 y 命令与光标移动命令共同作用的范例。

表 12-4　复制命令

命令	复制内容
yy	当前行
5yy	当前行和之后 4 行
yW	当前字符到下一单词的起始
y$	当前字符到当前行的末尾
y0	当前字符到当前行的起始
y^	当前字符到当前行下一个非空字符
yG	当前行到文件末尾
y20G	当前行到文件第 20 行

现在让我们来练习一下复制和粘贴。我们将光标移至文本的第一行，使用

yy 命令复制当前行。接下来，将光标移至最后一行（G），使用 p 命令将复制的内容粘贴到当前行的下方。

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
Line 5
The quick brown fox jumped over the lazy dog. It was cool.
```

命令 u 会取消我们的操作。将光标移至文件的最后一行，输入 p 命令将文本粘贴到当前行的上方。

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
The quick brown fox jumped over the lazy dog. It was cool.
Line 5
```

将表 12-4 中的其他命令都练习一下，以实际了解 p 命令和 P 命令的作用。练习结束后，将文件恢复到本来的样子。

12.6.5 合并行

vi 在行的概念上非常严格。通常来说，将光标移动到行的末端并删除行的末尾字符并不能将此行与下一行合并。因此，vi 专门提供了 J 命令（不要与移动光标的 j 命令混淆）来合并行。

若将光标置于第 3 行并输入 J 命令，将得到如下所示的结果。

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3 Line 4
Line 5
```

12.7 查找和替换

vi 提供了在一行或者整个文件中，根据搜索条件将光标移动至指定位置的功能。vi 还可以执行文本替换工作，用户可指定替换时是否需要用户确认。

12.7.1 行内搜索

命令 f 在行内进行搜索，并将光标移至搜索到的下一个指定字符。比如，命令 fa 就会将光标移动到本行下一处出现字符 a 的地方。在执行过一次行内搜索之后，

输入分号可以使 vi 重复上一次搜索。

12.7.2 搜索整个文件

同第 3 章中讲解过的 less 程序一样，命令“/”可以完成对单词或短语的搜索。当用户使用“/”命令后，一个“/”符号会出现在屏幕的底部。接下来，输入需要搜索的单词或短语，以 Enter 结束。光标就会移动到下一处包含被搜索字符串的地方。使用 n 命令可以重复此搜索。如下例所示。

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
Line 5
```

将光标移至文件的第一行，并输入如下代码。

```
/Line
```

输入 Enter 以结束，光标将移动至第 2 行。接下来，输入 n，光标将继续移动至第 3 行。重复输入 n 直至光标移动到文档的最后，且找不到符合条件的字符串。尽管现在只讲解到 vi 的单词和词组的搜索模式，但是 vi 同样支持正则表达式（一种强大的表达复杂文本模式的方法）的应用。第 19 章将会讲解这方面的内容。

12.7.3 全局搜索和替换

vi 使用 ex 命令来执行几行之内或者整个文件中的搜索和替换操作。输入以下命令可将文件中的 Line 替换为 line。

```
:%s/Line/line/g
```

现在就来解析这条命令每一部分的功能（见表 12-5）。

表 12-5 全局搜索和替换语法范例

组成	含义
:	分号用于启动一条 ex 命令
%	确定了操作作用的范围。%简洁地代表了从文件的第 1 行到最后 1 行。本命令的范围还可以表示为 1,5（因为本文件只有 5 行），或者是 1,$，意思是“从第 1 行到文件的最后一行”。如果不明确指出命令的作用范围，那么命令只会在当前行生效
s	指定了具体的操作——本次是替换操作（搜索和替换）

续表

组成	含义
/Line/line	搜索和替换的文本
g	代指 global（全局），也就是说对搜索到的每一行的每一个实例进行替换。如果 g 缺失，那么只替换每一行第一个符合条件的实例

以下是执行过查找和替换命令之后的文档内容。

```
The quick brown fox jumped over the lazy dog. It was cool.
line 2
line 3
line 4
line 5
```

在命令末尾添加 c，则命令在每次替换之前都会请求用户确认。如下所示。

```
:%s/line/Line/gc
```

此命令将会将文件替换回原来的样子，但是每次替换前，vi 都会停下来询问用户是否确认执行替换。

```
replace with Line (y/n/a/q/l/^E/^Y)?
```

圆括号中的每一个字符都是一种可能的回答，表 12-6 具体阐述了每一个字符的含义。

表 12-6　替换确认功能键

功能键	行为
y	执行替换
n	跳过此次替换
a	执行此次替换和之后的所有替换
q 或者 ESC	停止替换
l	执行此次替换并退出替换。是 last 的缩写
Ctrl-E，Ctrl-Y	分别是向下滚动和向上滚动，能用于查看替换处的上下文

12.8　编辑多个文件

用户经常遇到需要同时编辑多个文件的情况。可能是需要对多个文件作出修改，或者是拷贝文件的部分内容到另一个文件。用户可以通过在命令行具体指定多个文件的方式使 vi 打开多个文件。

```
vi file1 file2 file3...
```

现在退出所处的 vi 会话，并创建一个用于编辑的新文件。输入:wq 来退出 vi 并保存做出的修改。接下来，使用 ls 命令的部分输出在主目录创建一个用于实验的新文件。

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

现在就用 vi 来同时编辑旧文件和新文件。

```
[me@linuxbox ~]$ vi foo.txt ls-output.txt
```

vi 启动后，屏幕显示内容如下所示。

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
Line 5
```

12.8.1 切换文件

使用以下 ex 命令来从一个文件切换到下一个文件。

```
:n
```

切换回上一个文件。

```
:N
```

当用户从一个文件切换到另一个的时候，vi 要求用户必须先保存对当前文件做出的修改才能切换到其他文件。若要放弃对文件的修改并使 vi 强制切换到另一个文件，可在命令后添加感叹号（!）。

除了以上描述的切换方法之外，vim（和一些版本的 vi）还提供了一些 ex 命令让用户可以更轻松地编辑多个文本。用户可使用:buffers 命令来查看正在编辑的文件列表。

```
:buffers
  1 %a   "foo.txt"                    line 1
  2      "ls-output.txt"              line 0
Press ENTER or type command to continue
```

输入:buffer 加文件（buffer）编号可切换到另一个文件（buffer）。如从文件 1（foo.txt）切换到文件 2（ls-output.txt），用户应当输入如下命令。

```
:buffer 2
```

现在屏幕展示的就是文件 2 的内容了。

12.8.2 载入更多的文件

我们也可以在现有的编辑会话中载入更多的文件。使用 ex 命令:e（edit 的缩写）加文件名可以载入另一个文件。先退出现有的编辑会话并回到命令行模式。

重启 vi，并只打开一个文件。

```
[me@linuxbox ~]$ vi foo.txt
```

添加一个文件到编辑会话中，输入下列代码。

```
:e ls-output.txt
```

屏幕将展示第二个文件的内容，而第一个文件仍然处在编辑状态，可使用:buffers 命令来证实。

```
:buffers
  1 #    "foo.txt"                     line 1
  2 %a   "ls-output.txt"               line 0
Press ENTER or type command to continue
```

注意　使用:ez 载入的文件不会响应:n 或者:N 命令，而需使用:buffer 加文件编号来切换文件。

12.8.3 文件之间的内容复制

用户在编辑多个文件的过程中，有时会需要将一个文件中的一部分复制到另一个文件中。使用之前使用过的复制和粘贴命令即可完成此功能，示范如下。首先，在载入的两个文件中，切换到文件 1（foo.txt）。

```
:buffer 1
```

此时屏幕显示如下所示。

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
Line 5
```

接下来，将光标移动到文件的第一行并输入 yy 命令来复制第一行。

输入如下命令以切换到文件 2。

```
:buffer 2
```

现在屏幕将会展示一份文件列表，如下所示（这里只展示了一小部分）。

```
total 343700
-rwxr-xr-x 1 root root       31316 2011-12-05 08:58 [
-rwxr-xr-x 1 root root        8240 2011-12-09 13:39 411toppm
-rwxr-xr-x 1 root root      111276 2012-01-31 13:36 a2p
-rwxr-xr-x 1 root root       25368 2010-10-06 20:16 a52dec
-rwxr-xr-x 1 root root       11532 2011-05-04 17:43 aafire
-rwxr-xr-x 1 root root        7292 2011-05-04 17:43 aainfo
```

将光标移动到文件的第一行并使用 p 命令将从文件 1 复制的内容粘贴到本文件。

```
total 343700
The quick brown fox jumped over the lazy dog. It was cool.
-rwxr-xr-x 1 root root       31316 2011-12-05 08:58 [
-rwxr-xr-x 1 root root        8240 2011-12-09 13:39 411toppm
-rwxr-xr-x 1 root root      111276 2012-01-31 13:36 a2p
-rwxr-xr-x 1 root root       25368 2010-10-06 20:16 a52dec
-rwxr-xr-x 1 root root       11532 2011-05-04 17:43 aafire
-rwxr-xr-x 1 root root        7292 2011-05-04 17:43 aainfo
```

12.8.4 插入整个文件

用户还可以将一个文件完全插入正在编辑的文件中。为了实际演示这项功能，先结束现有的 vi 会话并重启 vi 的同时只打开一个文件。

```
[me@linuxbox ~]$ vi ls-output.txt
```

屏幕将再次显示一份文件列表。

```
total 343700
-rwxr-xr-x 1 root root       31316 2011-12-05 08:58 [
-rwxr-xr-x 1 root root        8240 2011-12-09 13:39 411toppm
-rwxr-xr-x 1 root root      111276 2012-01-31 13:36 a2p
-rwxr-xr-x 1 root root       25368 2010-10-06 20:16 a52dec
-rwxr-xr-x 1 root root       11532 2011-05-04 17:43 aafire
-rwxr-xr-x 1 root root        7292 2011-05-04 17:43 aainfo
```

将光标移动到文件的第三行并输入如下 ex 命令。

```
:r foo.txt
```

命令:r（read 的缩写）将指定的文件内容插入到光标位置之前。现在的屏幕显示如下所示。

```
total 343700
-rwxr-xr-x 1 root root       31316 2011-12-05 08:58 [
-rwxr-xr-x 1 root root        8240 2011-12-09 13:39 411toppm
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
```

```
Line 3
Line 4
Line 5
-rwxr-xr-x 1 root root     111276 2012-01-31 13:36 a2p
-rwxr-xr-x 1 root root      25368 2010-10-06 20:16 a52dec
-rwxr-xr-x 1 root root      11532 2011-05-04 17:43 aafire
-rwxr-xr-x 1 root root       7292 2011-05-04 17:43 aainfo
```

12.9 保存工作

就像其他功能一样，vi 提供了很多种方式来保存编辑过的文件。前面的章节已经介绍过用于此功能的 ex 命令:w，但是还有一些其他可用的方法。

在命令模式下，输入 ZZ 将保存当前文档并退出 vi。同样的，ex 命令:wq 组合了:w 和:q 这两个命令的功能，能够保存文件并退出 vi。

当命令:w 指定一个随意的文件名时，命令的功能就类似于“另存为”。例如，用户在编辑 foo.txt 的时候想要将其另存为 foo1.txt，那么就可以输入如下内容。

```
:w foo1.txt
```

注意 此命令在以新名称保存文件的同时，并不更改编辑中的原文件的名称。当用户继续编辑时，编辑的还是 foo.txt 而不是 foo1.txt。

第 13 章

定制提示符

本章将会讲解一个看似微不足道的细节：shell 提示符。通过讲解，我们会发现 shell 和终端仿真器程序的内部工作机制。

和 Linux 中的很多程序一样，shell 提示符的可配置性很高。尽管大多数用户并不重视提示符，但是，一旦我们学会了怎样控制它，它就会成为一种相当有用的设备。

13.1 提示符的分解

系统的默认提示符看起来如下所示。

```
[me@linuxbox ~]$
```

可以看到提示符中包含了用户名、主机名和当前的工作目录，但是为什么提示符是这个样子的呢？很简单，提示符就是这样定义的。提示符是由名为 PS1（prompt string 1 的缩写，即提示符字符串 1）的环境变量定义的。echo 命令可

以帮助用户看到 PS1 的值。

```
[me@linuxbox ~]$ echo $PS1
[\u@\h \W]\$
```

注意 如果输出的结果同本书的范例不同，也不需要担心。每一个 Linux 发行版本对此提示符字符串的定义都会有所不同，有一些甚至定义得很奇怪。

可以看出，PS1 包含了一些提示符中出现的符号，比如方括号、@符号和美元符号，但是其余的部分则很令人困惑。聪明的读者会将这些符号与表 7-2 中所示的由反斜杠转义的特殊字符联系起来。

表 13-1 shell 提示符中使用的转义字符

转义字符	含义
\a	ASCII 铃声。在遇到该转义字符时，计算机发出哔哔声
\d	当前日期，以星期、月、日的形式表示，如“Mon May 26”
\h	本地机器的主机名，但是不带域名
\H	完整的主机名
\j	当前 shell 会话中进行的任务个数
\l	当前终端设备的名称
\n	换行符
\r	回车符
\s	shell 程序的名称
\t	当前时间（24 小时制），格式为小时：分钟：秒
\T	当前时间（12 小时制）
\@	当前时间（12 小时制，格式为 AM/PM
\A	当前（24 小时制），格式为小时：分钟
\u	当前用户的用户名
\v	shell 的版本号
\V	shell 的版本号和发行号
\w	当前工作目录名
\W	当前工作目录名称的最后一部分
\!	当前命令的历史编号
\#	当前 shell 会话中输入的命令数
\$	在非管理员权限下输出“$”。在管理员权限下输出“#”
\[	标志一个或多个非打印字符序列的开始。用于嵌入非打印的控制字符，使其以一定方式操纵终端仿真器，比如移动光标或更改文本颜色
\]	标志着非显示字符序列的结束

13.2 尝试设计提示符

通过这个特殊字符列表，我们可以更改提示符来查看效果。我们首先备份现有的字符串，以便过后进行恢复。为此，将现有的字符串复制到我们创建的另外一个 shell 变量中。

```
[me@linuxbox ~]$ ps1_old="$PS1"
```

这样我们就创建了名为 ps1_old 的新变量，并将 PS1 的值赋给了 ps1_old。我们可以使用 echo 命令来验证 PS1 的值确实已经被复制了。

```
[me@linuxbox ~]$ echo $ps1_old
[\u@\h \W]\$
```

在终端会话中，用户随时可以通过这个过程的逆操作来复原最初的提示符。

```
[me@linuxbox ~]$ PS1="$ps1_old"
```

现在一切准备就绪。接下来让我们看看如果提示符为空会发生什么。

```
[me@linuxbox ~]$ PS1=
```

若提示符为空，那么用户不会得到任何提示。根本就没有提示字符串嘛！尽管提示符就在那里，但是系统并不会显示。这样的提示看起来很令人困惑，所以现在将提示符设置为最简略的内容。

```
PS1="\$ "
```

这样就好多了，至少现在用户知道自己在做什么了。可以注意到双引号中末尾的空格。当显示提示符时，这个空格会把美元符号和光标分隔开。

在提示符中添加一个铃声。

```
$ PS1="\a\$ "
```

这样以来，每当系统显示提示符的时候，用户都会听到哔哔声。虽然这可能会使用户感到厌烦，但是在一些情况下可能会很有帮助，比如可以在一个耗时特别长的命令执行完毕时通知用户。

接下来，我们试着创建一个信息丰富的提示符，其中包括主机名和当天的时间信息。

```
$ PS1="\A \h \$ "
17:33 linuxbox $
```

如果我们需要记录某些任务的执行时间，在提示符中添加时间信息会比较有用。最后，我们定制一个类似于最初样式的提示符。

```
17:37 linuxbox $ PS1="<\u@\h \W>\$ "
<me@linuxbox ~>$
```

用户可以尝试使用表 13-1 中其他的序列，看看能不能创造出一个奇妙的新提示符。

13.3 添加颜色

大多数终端都会响应某些非打印字符序列，来控制光标位置、字符属性（如颜色、粗体、文本闪烁等）等内容。本章稍后会讲解光标位置，现在先来讲解颜色。

混乱的终端

很久以前，当终端机与远程计算机还紧密联系在一起的时候，我们有很多种不同品牌的终端机，并且每一种都以不同的方式工作。这些终端的键盘不同，对控制信息的诠释方式也不同。UNIX 和类 UNIX 系统都配备有两种非常复杂的子系统（分别叫做 termcap 和 terminfo）来处理终端控制领域的混乱局面。如果你查看一下终端仿真器最底层的属性设置，可能会找到一个关于终端仿真器类型的设置。

为了使所有的终端都使用同一种通用语言，美国国家标准委员会（ANSI）开发了一套标准的字符序列，来控制视频终端。使用过 DOS 的老用户一定会记得用来启用这些代码解释的 ANSI.SYS 文件。

字符颜色是由发送到终端仿真器的一个 ANSI 转义代码来控制的，该转义代码嵌入到了要显示的字符流中。控制代码不会“打印”到屏幕上，而是被被终端解释为一条指令。在表 13-1 中可以看到，“\[”和“\]”这两个序列用来封装非打印字符串。一个 ANSI 转义代码以八进制 033（该代码由转义键[escape key]产生）开始，后面跟着一个可选的字符属性，之后是一条指令。例如，将文本颜色设置为正常（attribute = 0）、黑色的代码是\033[0;30m。

表 13-2 列出了可用的文本颜色。需要注意的是，这些颜色分为两组，区别在于是否应用了粗体（bold）属性（1），这个属性使得色彩分为深色和浅色。

表 13-2 设置文本颜色的转义序列

字符序列	文本颜色
\033[0;30m	黑色
\033[0;31m	红色
\033[0;32m	绿色
\033[0;33m	棕色
\033[0;34m	蓝色
\033[0;35m	紫色
\033[0;36m	青色
\033[0;37m	淡灰色
\033[1;30m	深灰色
\033[1;31m	淡红色
\033[1;32m	淡绿色
\033[1;33m	黄色
\033[1;34m	淡蓝色
\033[1;35m	淡紫色
\033[1;36m	淡青色
\033[1;37m	白色

现在让我们尝试创造红色的提示符（本书中表现为灰色）。我们将相应的转义代码插入提示符的开端。

```
<me@linuxbox ~>$ PS1="\[\033[0;31m\]<\u@\h \W>\$ "
<me@linuxbox ~>$
```

事实证明操作可行，但是此时用户输入的所有文字也变成红色了。要修复这个问题，可以在提示符的末尾插入另一条转义码，以通知终端仿真器恢复到原来的颜色。

```
<me@linuxbox ~>$ PS1="\[\033[0;31m\]<\u@\h \W>\$\[\033[0m\] "
<me@linuxbox ~>$
```

这样就好多了。

使用表 13-3 中的代码可以设置文本的背景颜色，背景颜色不支持粗体属性。

表 13-3 设置背景颜色的转义序列

字符序列	背景颜色
\033[0;40m	黑色
\033[0;41m	红色

续表

字符序列	背景颜色
\033[0;42m	绿色
\033[0;43m	棕色
\033[0;44m	蓝色
\033[0;45m	紫色
\033[0;46m	青色
\033[0;47m	淡灰色

通过为第一个转义代码做一些修改，就可以创建带有红色背景的提示符。

```
<me@linuxbox ~>$ PS1="\[\033[0;41m\]<\u@\h \W>\$\[\033[0m\] "
<me@linuxbox ~>$
```

用户可以尝试使用其他颜色代码，看看分别可以创造出什么样的提示符。

注意 文本除了正常（0）和粗体（1）属性外，还可以设置为下划线（4）、闪烁（5）和斜体（7）。为了维持好的品味，许多终端仿真器拒绝使用闪烁属性。

13.4 移动光标

转义代码也可以用来定位光标。比如在提示符出现的时候，这些转义代码通常用来在屏幕的不同位置（比如屏幕上方的一角）显示一个时钟或其他信息。表 13-4 所示为可以定位光标的转义代码。

表 13-4 光标移动转义序列

转义码	动作
\033[l;cH	将光标移动至 l 行 c 列
\033[nA	将光标向上移动 n 行
\033[nB	将光标向下移动 n 行
\033[nC	将光标向前移动 n 个字符
\033[nD	将光标向后移动 n 个字符
\033[2J	清空屏幕并将光标移动至左上角（第 0 行第 0 列）
\033[K	清空当前光标位置到行末的内容
\033[s	存储当前光标位置
\033[u	恢复之前存储的光标位置

通过使用这些代码，用户可以构建这样的一条提示符。每当提示符出现时，屏幕的上方会绘制出一个红色的横条，横条中有用黄色文本显示的时间。用于

提示符的编码就是一个看起来很可怕的字符串：

```
PS1="\[\033[s\033[0;0H\033[0;41m\033[K\033[1;33m\t\033[0m\033[u\]<\u@\h \W>\$ "
```

表 13-5 分析了这个字符串中每一部分的作用。

表 13-5　复杂提示符的分解

字符序列	动作
\[	开始一个非打印字符序列。其真正目的是为了让 bash 正确计算可见提示符的长度。如果没有该字符，命令行编辑功能无法正确定位光标
\033[s	存储光标位置。在屏幕的顶部横条绘制完成并显示时间后，读取并使光标返回此位置。需要注意的是，一些终端仿真器不支持该代码
\033[0;0H	将光标移动至左上角，即第 0 行第 0 列
\033[0;41m	将背景颜色设置为红色
\033[K	将光标当前位置（左上角）到行末的内容清空。因为现在背景颜色已经是红色了，所以清空后的行就是红色，也就绘制出了红色的横条。需要注意的是，清空行的内容并不会改变光标的位置，光标仍处于屏幕左上角
\033[1;33m	将文本颜色设置为黄色
\t	显示当前时间。尽管这是一个可打印的元素，但是还是将其包含在提示符非打印部分中，这是因为 bash 在计算可见提示符的长度时，不应当将其计算在内
\033[0m	关闭颜色。对文本和背景均有效
\033[u	恢复之前存储的光标位置
\]	结束非打印的字符序列
<\u@\h \W>\$	提示符字符串

13.5 保存提示符

很显然，用户不会想要每次都输入这样一长串代码，所以就需要将提示符存储在某个地方。将提示符添加到.bashrc 文件中是一个一劳永逸的解决办法，也就是将以下两行代码添加到文件中。

```
PS1="\[\033[s\033[0;0H\033[0;41m\033[K\033[1;33m\t\033[0m\033[u\]<\u@\h \W>\$ "
export PS1
```

13.6 本章结尾语

无论你信是不信，还有很多事情也可以通过提示符来完成，这会涉及我们这里没有讲解到的 shell 函数和脚本，但这是一个好的开始。因为默认的提示符通常已经能让用户满意，所以并不是每个人都会想要对提示符做出修改。但是对于那些喜欢探索改进的用户来说，shell 提示符提供了很多制造琐碎乐趣的机会。

第三部分

常见任务和主要工具

第 14 章 软件包管理

如果用户经常访问 Linux 社区，那么针对众多 Linux 发行版本中哪一版是最好的这一问题，一定听到过诸多观点。通常，有关此问题的讨论都显得非常无聊，因为他们都将侧重点放在诸如哪个版本的桌面背景最漂亮（有些人居然因为 Ubuntu 的默认配色方案而不选择使用它）以及其他鸡毛蒜皮的小事上。

其实，决定 Linux 发行版本质量最重要的因素是软件包系统和支持该发行版本社区的活力。进一步接触 Linux，我们就会发现 Linux 软件的研究现状相当活跃。事物总是在不断变化，许多一流的 Linux 发行版本每 6 个月就有一个新版本问世，而且许多个人程序每天都在更新。要想同步这些日新月异的软件，我们就需要好的工具进行软件包管理。

软件包管理是一种在系统上安装、维护软件的方法。目前，很多人通过安装 Linux 经销商发布的软件包来满足他们所有的软件需求。这与早期的 Linux 形成了鲜明的对比。因为在 Linux 早期，想要安装软件必须先下载源代码，然后对其进行编译。这并不是说编译源代码不好，源代码公开恰是 Linux 吸引人

的一大亮点。编译源代码赋予用户自主检查、提升系统的能力，只是使用预先编译的软件包会更快、更容易些。

本章会介绍一些用于 Linux 软件包管理的命令行工具。虽然所有主流的 Linux 发行版本都提供了强大而复杂的维持系统运行的图形化界面操作程序，但学习命令行程序同样重要，因为它可以执行许多图形化程序很难甚至无法完成的任务。

14.1 软件包系统

不同的 Linux 发行版用的是不同的软件包系统，并且原则上，适用于一种发行版的软件包与其他版本是不兼容的。多数 Linux 发行版采用的不外乎两种软件包技术阵营，即 Debian 的.deb 技术和 Red Hat 的.rpm 技术。当然也有一些特例，比如 Gentoo、Slackware 和 Foresight 等，但多数版本采取的还是表 14-1 中所列的两个基本软件包系统。

表 14-1 主流软件包系统类

软件包系统	发行版本（只列举了部分）
Debian 类（.deb 技术）	Debian、Ubuntu、Xandros、Linspire
Red Hat 类（.rpm 技术）	Fedora、CentOS、Red Hat Enterprise Linux、openSUSE、Mandriva、PCLinuxOS

14.2 软件包系统工作方式

在非开源软件产业中，给系统安装一个新应用，通常需先购买“安装光盘”之类的安装介质，然后运行安装向导进行安装。

Linux 并不是这样。事实上，Linux 系统所有软件均可在网上找到，并且多数是以软件包文件的形式由发行商提供，其余则以可手动安装的源代码形式存在。本书第 23 章将会简述如何通过编译源代码安装软件。

14.2.1 软件包文件

包文件是组成软件包系统的基本软件单元，它是由组成软件包的文件压缩而成的文件集。一个包可能包含大量的程序以及支持这些程序的数据文件，包文件既包含了安装文件，又包含了有关包自身及其内容的文本说明之类的软件包

元数据。此外，许多软件包中还包含了安装软件包前后执行配置任务的安装脚本。

包文件通常由软件包维护者创建，该维护者通常（并不总是）是发行商的职员。包维护者从上游供应商（一般是程序的作者）获得软件源代码，然后进行编译，并创建包的元数据及其他必需的安装脚本。通常，包维护者会在初始源代码上做部分修改，从而提高该软件包与该 Linux 发行版本其他部分的兼容性。

14.2.2 库

虽然一些软件项目选择自己包装和分销，但如今多数软件包均由发行商或感兴趣的第三方创建。Linux 用户可以从其所使用的 Linux 版本的中心库中获得软件包。所谓的中心库，一般包含了成千上万个软件包，而且每一个都是专门为该发行版本建立和维护的。

在软件开发生命周期的不同阶段，一个发行版本可能会维护多个不同仓库。例如，通常会有一个测试库，该库里面存放的是刚创建的、用于调试者在软件包正式发布前查找漏洞的软件包。另外，一个发行版本通常还会有一个开发库，存放的是下一个公开发行的版本中所包含的开发中的软件包。

一个发行版本可能还会有相关的第三方库，这些库通常提供因法律原因，如专利或数字版权管理（DRM）等反规避问题而不能包括在发行版本中的软件，著名实例就是加密的 DVD 技术支持，该做法在美国不合法。第三方库主要用在软件专利和反规避法不适用的国家，这些库通常完全独立于它们所支持的 Linux 版本，用户必须充分了解后手动将其加入到软件包文件管理系统的配置文件中，才能使用它们。

14.2.3 依赖关系

几乎没有任何一个程序是独立的。与之相反，程序之间相互依赖彼此完成既定工作。一些共有的操作，比如输入/输出操作，就是由多个程序共享的例程执行。这些例程存储在共享库里面，共享库里面的文件为多个程序提供必要的服务。如果一个软件包需要共享库之类的共享资源，说明其具有依赖性。现代软件包管理系统都提供依赖性解决策略，从而确保用户安装了软件包的同时也安装了其所有的依赖关系。

14.2.4 高级和低级软件包工具

软件包管理系统通常包含两类工具——执行如安装、删除软件包文件等任

务的低级工具和进行元数据搜索及提供依赖性解决的高级工具。本章将要介绍 Debian 类型的系统（如 Ubuntu 等类似系统）所提供的软件包工具和最近的 Red-Hat 系列产品使用的工具。尽管所有 Red-Hat 系列版本都使用相同的低级工具（rpm），但使用的高级工具却不尽不同。下面我们将讨论高级软件包工具 yum 程序，它为高级 Fedora、Red Hat Enterprise Linux（红帽企业版 Linux）和 CentOS 等系统所用，而其他 Red Hat 系列的发行版本也提供功能与之相媲美的高级工具，具体见表 14-2。

表 14-2 软件包系统工具

发行版本	低级工具	高级工具
Debian 类	dpkg	apt-get、aptitude
Fedora、Red Hat Enterprise Linux、 CentOS	rpm	yum

14.3 常见软件包管理任务

命令行软件包管理工具可以完成许多操作，下面我们介绍一些较常见的。有一点要说明，低级工具也支持软件包文件的创建，但不在本书的讨论范围。

在下面的讨论中，单词 package_name 指软件包的实际名称，而 package_file 则是指包含该软件包的文件名。

14.3.1 在库里面查找软件包

通过使用高级工具来搜索库元数据时，我们可以根据包文件名或其描述来查找该包。

表 14-3 包搜索命令

系统类型	命令
Debian 系统	apt-get update apt-cache search search_string
Red Hat 系统	yum search search_string

例如，在 Red Hat 系统的 yum 库中搜索 emac 文本编辑器的代码如下。

```
yum search emacs
```

14.3.2 安装库中的软件包

高级工具允许从库中下载、安装软件包，同时安装所有的依赖包（见表

14-4）。

表 14-4 软件包安装命令

系统类型	命令行
Debian 系统	apt-get update apt-get install package_name
Red Hat 系统	yum install package_name

例如，在 Debian 系统上安装 apt 元数据库中的 emac 文本编辑器的代码如下。

```
apt-get update; apt-get install emacs
```

14.3.3 安装软件包文件中的软件包

如果软件包文件并不是从库源中下载的，那么我们就可用低级工具直接安装（但并不安装依赖性关系），具体见表 14-5。

表 14-5 低级软件包安装命令

系统类型	命令
Debian 系统	dpkg --install package_file
Red Hat 系统	rpm -i package_file

例如，当 emacs-22.1-7.fc7-i386.rpm 软件包文件从非库资源网站下载时，可采用如下方式安装于 Red Hat 系统中。

```
rpm -i emacs-22.1-7.fc7-i386.rpm
```

注意 由于该方法采用低级 rpm 工具安装，所以并不会解决依赖性关系。一旦 rpm 在安装过程中发现缺少依赖包，rpm 就会跳出错误后退出。

14.3.4 删除软件包

卸载软件包既可利用高级工具也可用低级工具，高级工具的相关命令见表 14-6。

表 14-6 软件包移除命令

系统类型	命令
Debian 系统	apt-get remove package_name
Red Hat 系统	yum erase package_name

例如，从 Debian 系统中卸载 emacs 软件包的代码如下。

```
apt-get remove emacs
```

14.3.5 更新库中的软件包

最常见的软件包管理任务是保持系统安装最新的软件包。高级工具仅需要一步便可完成此重要任务（见表 14-7）。

表 14-7 软件包更新命令

系统类型	命令
Debian 系统	apt-get update; apt-get upgrade
Red Hat 系统	yum update

例如，更新所有已安装于 Debian 系统中的可更新软件包的代码如下。

```
apt-get update; apt-get upgrade
```

14.3.6 更新软件包文件中的软件包

如果软件包的更新版本已从非库源中下载，那么我就可以用表 14-8 所列的命令进行安装更新从而取代原版本。

表 14-8 低级软件包更新命令

系统类型	命令
Debian 系统	dpkg --install package_file
Red Hat 系统	rpm -U package_file

例如，将 Red Hat 系统上已安装好的 emac 程序更新为 emacs-22.1-7.fc7-i386.rpm 软件包文件中的版本的代码如下。

```
rpm -U emacs-22.1-7.fc7-i386.rpm
```

注意　与 rpm 命令不同，dpkg 命令在更新软件包时并没有指定的参数选项，只有在安装软件包时才有。

14.3.7 列出已安装的软件包列表

表 14-9 中所列出的命令用于显示系统上所有已安装的软件包列表。

表 14-9 软件包列表命令

系统类型	命令
Debian 系统	dpkg --list
Red Hat 系统	rpm -qa

14.3.8 判断软件包是否安装

表 14-10 中所列的为低级工具用于判断系统是否已安装某个软件的命令。

表 14-10 软件包状态命令

系统类型	命令
Debian 系统	dpkg --status package_name
Red Hat 系统	rpm -q package_name

例如，判断 emac 程序包在 Debian 系统中是否已安装的代码如下。

```
dpkg --status emacs
```

14.3.9 显示已安装软件包的相关信息

在已知已安装的软件包的名称的情况下，便可以用表 14-11 中的命令显示该软件包的描述信息。

表 14-11 软件包信息查看命令

系统类型	命令
Debian 系统	apt-cache show package_name
Red Hat 系统	yum info package_name

例如，查看 Debian 系统上 emac 软件包的描述信息的代码如下。

```
apt-cache show emacs
```

14.3.10 查看某具体文件由哪个软件包安装得到

表 14-12 中的命令判断某个特定的文件是由哪个软件包负责安装的。

表 14-12 查询文件所属命令

系统类型	命令
Debian 系统	dpkg --search file_name
Red Hat 系统	rpm -qf file_name

例如，查看 Red Hat 系统中哪个软件包安装了/usr/bin/vim 目录下的文件的代码如下。

```
rpm -qf /usr/bin/vim
```

14.4 本章结尾语

在后面的章节，我们将会讨论许多应用非常广泛的程序。尽管其中多数程序由系统默认安装，但有时仍然需要我们安装一些额外的软件包。就本章中介绍的软件包管理方面的知识，我们足以安装和管理以后所需要的程序。

Linux 软件安装的神秘之处

学习使用 Linux 平台的人有时会觉得在该平台上安装软件很困难，而且不同的发行版本所用的多样化软件包策略对使用者来说也是一大障碍。确实，这是屏障，但也只是对于那些希望发布私有软件二进制版本的专有软件厂商而言。

Linux 软件基于开放源代码的思想上。当程序的开发者发布了某产品的源代码后，很有可能会有一个与之相关的人打包该产品，并将其添加到该发行版的库中。这种方法可以确保该产品能够与发行版本保持很好的兼容性并且给使用者提供一个一站式的软件购买平台，而不需要搜索每个产品的网站主页。

除了那些已经加入到 Linux 内核的、不再是库中独立个体的设备驱动，其他设备驱动的处理方式都差不多。总体而言，Linux 世界中并没有“驱动光盘”这一类东西。Linux 系统很干脆，其内核要么支持该设备要么不支持。事实上，Linux 内核要比 Windows 内核支持的设备多很多。当然，如果你所需要的特定设备不被 Linux 支持，那即便比 Windows 多也无济于事。如果发生这样的情况，便需要寻找原因。设备不被支持一般是由以下三个原因造成的。

- **设备太新**。由于许多硬件开发商并不积极支持 Linux 的开发，这就需要 Linux 社区的成员花时间编写该硬件设备的驱动代码。
- **设备太稀少**。并不是所有的发行版本都包括了所有的设备驱动。每一个发行版本都建立了自己的内核，并且由于内核本身是可配置的（正是由于内核可配置，所以 Linux 才能在运行在从手表到大型主机的所有设备上），所以发行版本可能就会忽略了某个特定的设备。通过寻找以及下载该设备

的源代码，你完全可以自己编译、安装该驱动。这个过程虽不是那么困难，但却有点复杂。本书第 23 章将会讨论如何编译软件。

- **硬件供应商隐藏了某些东西**。某些硬件既没有发布 Linux 驱动的源代码，也没有提供给他人编写驱动代码的技术文档，也就是说该硬件供应商打算保密此硬件编程接口。一般来说，使用者都不希望自己的计算机里有保密设备，所以建议移除这些硬件，并将其与其他无用文件一并放入垃圾箱。

第15章 存储介质

前面的章节我们主要讨论了文件级别的数据处理，本章我们将会讨论设备级别的数据处理。对于诸如硬盘之类的物理存储器、网络存储器以及像 RAID（独立冗余磁盘阵列）和 LVM（逻辑卷管理）之类的虚拟存储器，Linux 都具有惊人的处理能力。

然而，本书并不是以介绍系统管理为主，所以在此我们并不打算深入探讨这个主题，我们只会简单介绍其基本概念以及用于管理存储设备的一些重要命令。

为了能够更好地练习本章中的例子，我们需要使用一个 USB 闪存、一张 CD-RW 光盘（用于可进行光盘刻录的系统）以及一张软盘（如果系统就是这么配置）。

本章将会介绍如下命令。

- mount：挂载文件系统。
- unmount：卸载文件系统。

- fdisk：硬盘分区命令。
- fsck：检查修复文件系统。
- fdformat：格式化软盘。
- mkf：创建文件系统。
- dd：向设备直接写入面向块数据。
- genisoimage（mkisofs）：创建一个 ISO 9600 映像文件。
- wodim（cdrecord）：向光存储介质写入数据。
- md5sum：计算 MD5 校验码。

15.1 挂载、卸载存储设备

Linux 图形界面操作最近所取得的进展已使得图形界面操作用户能非常容易地管理存储设备。多数情况下，设备只要连接上系统就能运作。但是，过去（差不多在 2004 年），这必须通过手动操作。由于像服务器这类的非图形界面操作系统通常都有一些极致的存储需求和复杂的配置要求，所以在这类系统中管理存储设备很大程度上还是靠手动操作。

管理存储设备首先要做的就是将该设备添加到文件系统树中，从而允许操作系统可以操作该设备，这个过程称之为挂载。回忆一下第 2 章所讲的知识，类 UNIX 操作系统，与 Linux 相似，都只有一个文件系统树，设备则都连接到树的不同点上。这就与其他操作系统诸如 MS-DOS、Windows 等不同，它们对于每个设备都有独立的树（如 C:\、D:\等）。

/etc/fstab 文件内容列出了系统启动时挂载的设备（通常是硬盘分区）。下例所示的是某个 Fedora 7 系统上/etc/fstab 文件的内容。

```
LABEL=/12         /          ext3     defaults          1 1
LABEL=/home       /home      ext3     defaults          1 2
LABEL=/boot       /boot      ext3     defaults          1 2
tmpfs             /dev/shm   tmpfs    defaults          0 0
devpts            /dev/pts   devpts   gid=5,mode=620    0 0
sysfs             /sys       sysfs    defaults          0 0
proc              /proc      proc     defaults          0 0
LABEL=SWAP-sda3   swap       swap     defaults          0 0
```

此文件中列出的文件系统多数是虚拟的，不适用于当前的讨论。其前三项内容，才是我们所要关注的重点内容。

```
LABEL=/12       /          ext3    defaults    1 1
LABEL=/home     /home      ext3    defaults    1 2
LABEL=/boot     /boot      ext3    defaults    1 2
```

这三行内容其实指的是硬盘分区，文件中的每一行含 6 个字段，如表 15-1 所示。

表 15-1　/etc/fstab 文件 6 个参数含义

字段	内容	描述
1	设备	通常，该字段表示的是与物理设备相关的设备文件的真实名称，比如 dev/hda1 就代表第一个 IDE 通道上的主设备的第一块分区。但是如今的计算机都有很多可热拔插的设备（像 USB 驱动器），所以许多较新的 Linux 发行版用文本标签来关联设备。当设备与系统连接后，该标签（格式化后就会加到存储介质中）就会被操作系统识别。通过这样的方式，不管实际的物理设备被分配到哪个设备文件，它仍然能被正确识别
2	挂载节点	设备附加到文件系统树上的目录
3	文件系统类型	Linux 能够挂载许多文件系统类型，最常见的原始文件系统是 ext3，但也支持许多其他系统如 FAT16（msdos）、FAT32（vfat）、NTFS（ntfs）、CD-ROM（iso9660）等
4	选项	文件系统挂载时可以使用许多选项参数，比如，可以设置文件系统以只读的方式挂载或是阻止任何程序修改它们（对于可移动设备是一个很有用的维护安全性的方法）。
5	频率	此数值被 dump 命令用来决定是否对该文件系统进行备份以及多久备份一次
6	优先级	此数值被 fsck 命令用来决定在启动时需要被扫描的文件系统的顺序

15.1.1　查看已挂载的文件系统列表

mount 命令用于文件系统挂载。不带任何参数输入该命令将会调出目前已经挂载的文件系统列表：

```
[me@linuxbox ~]$ mount
/dev/sda2 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/sda5 on /home type ext3 (rw)
/dev/sda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
fusectl on /sys/fs/fuse/connections type fusectl (rw)
/dev/sdd1 on /media/disk type vfat (rw,nosuid,nodev,noatime,
uhelper=hal,uid=500,utf8,shortname=lower)
twin4:/musicbox on /misc/musicbox type nfs4 (rw,addr=192.168.1.4)
```

列表的格式是：*device* on *mont_point* type *filesystem_type* (*options*)。例如，

上例中的第一行表示 dev/sda2 设备挂载在根目录下，可读写（后面的参数选项是 rw），属于 ext3 类型。然而，可以看到该列表的末尾有两个有趣的条目，倒数第二个条目表示/media/disk 目录下挂载了读卡器中 2GB 的 SD 记忆卡，最后一个条目表示/misc/musicbox 目录下挂载了一个网络驱动。

以 CD-ROM 为例，在我们插入 CD-ROM 前，首先查看一下系统信息：

```
[me@linuxbox ~]$ mount
/dev/mapper/VolGroup00-LogVol00 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/hda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
```

该列表来自于一个使用 LVM 机制创建其根文件系统的 CentOS5 系统。与许多现代 Linux 发行版一样，此系统在 CD-ROM 插入后会自动进行挂载。光盘插入后，输入 mount 命令，便会显示如下系统信息：

```
[me@linuxbox ~]$ mount
/dev/mapper/VolGroup00-LogVol00 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/hda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
/dev/hdc on /media/live-1.0.10-8 type iso9660 (ro,noexec,nosuid,nodev,uid=500)
```

与之前的信息列表相比，本列表只是在末尾处多了一个额外的条目，该条目表示 CD-ROM（本系统上的设备名是/dev/hdc）已经挂载在了/media/live-1.0.10-8 目录下并且是 iso9660 类型。此处只需要关注设备名，读者进行实验时，设备名很有可能就不一样。

警告 下面的例子中，要时刻注意自己使用的系统上显示的实际设备名，千万不要直接用本文中使用的名字。

同时，要注意音频 CD 与 CD-ROM 是不一样的。音频 CD 并不包含文件系统，所以通常意义上讲，音频 CD 不能被挂载。

获取 CD-ROM 的设备名之后，便可以卸载该设备，然后将其挂载在文件系统树的另外一个节点上。进行此操作，必须首先获得超级用户（使用适用于自己系统的命令切换为超级用户）权限，再使用 umount（注意拼写）命令卸载光盘。

```
[me@linuxbox ~]$ su -
Password:
[root@linuxbox ~]# umount /dev/hdc
```

接下来，我们为光盘创建一个新的挂载节点。挂载节点仅仅是文件系统上的某个目录，并没有什么特别之处，甚至都不需要是空目录，尽管如果在非空目录上挂载设备，该目录下原有内容将不可见直到此设备被卸载。作为演示，我们先创建一个新目录：

```
[root@linuxbox ~]# mkdir /mnt/cdrom
```

终于， CD 光盘挂载在了新的节点上，使用-t 选项指定文件系统类型：

```
[root@linuxbox ~]# mount -t iso9660 /dev/hdc /mnt/cdrom
```

之后，便可以通过新建的挂载节点访问 CD 光盘的内容：

```
[root@linuxbox ~]# cd /mnt/cdrom
[root@linuxbox cdrom]# ls
```

请注意，如果此时试图卸载 CD 光盘就会出现下面的问题：

```
[root@linuxbox cdrom]# umount /dev/hdc
umount: /mnt/cdrom: device is busy
```

为什么会出现这个问题？因为设备正在被某人或是某程序使用时是不能被卸载的。本例中的工作目录正好是 CD 光盘的挂载节点，所以导致了“设备繁忙”的错误警告。只要我们将工作目录改到挂载节点以外的地方就能轻松解决：

```
[root@linuxbox cdrom]# cd
[root@linuxbox ~]# umount /dev/hdc
```

如此该设备便卸载成功了。

为什么卸载如此重要

free 命令会输出关于存储器使用情况的一些数据，buffer（缓存）就包括在其中。计算机系统是以运行得尽可能快为原则设计的，阻碍计算机运行速度的一大因素就是低速设备。打印机则是一个典型的低速设备，从计算机的角度来看，即便是最快的打印机也已经是极其慢了。如果计算机必须停下来等待打印机完成一页的打印，那么计算机肯定会运行得相当慢。计算机出现早期，也就是还没有能够进行多任务处理的时期，这的确是个问题。每次打印电子表格或是文本文档时计算机就必须停止工作。计算机以打印机能够接受的最快速度向打印机传送数据，但是由于打印的速度很慢所以数据传送也很慢。打印缓冲区的问世，解决了数据传输慢的问题。打印缓冲区是存在于

计算机与打印机之间的 RAM 存储器设备，有了打印缓冲区，计算机就可以将准备发送给打印机的数据先发送给缓冲区，这个过程可以进行得很快因为 RAM 的存储速度很快，从而计算机能尽快返回去处理其他进程而不是停下来等待。与此同时，打印缓冲区再慢慢地以打印机能接受的速度从缓冲区存储中向打印机传送数据。

为了提升计算机的运行速度，缓冲区思想在计算机中得到了广泛运用。从/向较慢设备偶然性的读/写数据不再会妨碍系统的运行速度，操作系统会将那些已经从存储设备读取或是准备向存储设备写入的数据一直储存在内存中，直到这些数据确实需要与较慢设备发生交互。以 linux 系统为例，它似乎总是试图填满其内存，但这并不意味着 Linux 会用尽所有的存储器，只能说 Linux 正充分利用其可利用的一切内存进行尽可能多的缓存操作。

由于缓冲区的存在，向物理设备的写入操作被推迟到了未来某个时间，所以能快速向存储设备写入数据。与此同时，准备写入设备的数据不断向缓冲区堆积，而操作系统则不时地将这些数据写入物理设备。

卸载设备能确保缓存中的所有剩余数据全部写入设备，从而设备能被安全移除。如果设备事先没有卸载就被移除，那么缓存中就可能仍有剩余数据。有些情况下，这些未传输完的数据可能包含重要的目录更新信息，而这些信息会导致文件系统损坏，这时情况就糟透了。

15.1.2 确定设备名称

对于现在的系统，有时设备名称的确定比较困难。但是在过去，由于设备总是在一个地方固定不动，所以确定设备名并不是那么困难。类 UNIX 系统则偏爱这种方式，UNIX 开发初期，更改磁盘驱动就像用铲车将洗衣机大小的设备从机房间里面移除那样困难。近些年，典型台式机的硬件配置已经变得很灵活，而 Linux 也通过不断完善变得灵活了许多。

上例利用了现代 Linux 桌面系统的一种能力——自动挂载设备后确定设备名。但是如果操作的是一台服务器或者在不支持这样的自动挂载操作的情况下，我们那该怎么办？

要解决上述问题，让我们首先了解系统是如何命名设备的。查看/dev 目录（所有设备所在的目录）下的设备信息，我们会发现有海量的设备：

```
[me@linuxbox ~]$ ls /dev
```

ls 命令输出的表单内容揭示了设备命名的一些固定模式，表 15-2 列出了部分：

表 15-2　Linux 存储设备名称

模式	设备
/dev/fd*	软盘驱动器
/dev/hd*	较旧系统上的 IDE（或 PATA）硬盘。典型的主板有两个 IDE 连接点或通道，并且每个都有两个驱动器附着点。线缆上第一个驱动器叫做主设备，第二个叫做从设备。设备命名按照如下规则进行：/dev/hda 代表第一个通道上的主设备，/dev/hdb 代表第一个通道上的从设备；/dev/hdc 代表第二个通道上的主设备，以此类推，而末尾的数字代表设备的分区号。例如，当/dev/hda 代表整个硬盘时，/dev/hda1 表示该硬盘驱动上的第一块分区
/dev/lp*	打印机设备
/dev/sd*	SCSI 硬盘，在最近的 Linux 系统上，内核把所有的类硬盘设备（包括 PATA/SATA 硬盘、闪存、USB 海量存储设备比如便携式音乐播放器或数码相机等）都当作 SCSI 硬盘。剩下的命名规则与上面所讲的/dev/hd*的命名规则类似
/dev/sr*	光驱（CD/DV 播放机和刻录机）

另外，我们经常能看到像/dev/cdrom、/dev/dvd、/dev/floppy 这样的符号链接，它们都是指向实际设备文件的，使用符号链接只是为了使用方便。

如果读者使用的系统不能自动挂载可移动设备，那么可以用下面介绍的方法来命名插入系统的可移动设备。首先，对/var/log/messages 文件进行实时查看（此操作可能需要超级用户权利）：

```
[me@linuxbox ~]$ sudo tail -f /var/log/messages
```

文件的最后几行输出显示后停止该程序，接着插入可移动设备。以 16MB 的闪存为例，几乎在插入瞬间，内核就注意并且检测到了此设备：

```
Jul 23 10:07:53 linuxbox kernel: usb 3-2: new full speed USB device using uhci_h
cd and address 2
Jul 23 10:07:53 linuxbox kernel: usb 3-2: configuration #1 chosen from 1 choice
Jul 23 10:07:53 linuxbox kernel: scsi3 : SCSI emulation for USB Mass Storage dev
ices
Jul 23 10:07:58 linuxbox kernel: scsi scan: INQUIRY result too short (5), using
36
Jul 23 10:07:58 linuxbox kernel: scsi 3:0:0:0: Direct-Access Easy Disk 1.00 PQ:
0 ANSI: 2
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] 31263 512-byte hardware secto
rs (16 MB)
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Write Protect is off
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Assuming drive cache: write t
hrough
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] 31263 512-byte hardware secto
rs (16 MB)
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Write Protect is off
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Assuming drive cache: write t
hrough
Jul 23 10:07:59 linuxbox kernel:  sdb: sdb1
```

```
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Attached SCSI removable disk
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: Attached scsi generic sg3 type 0
```

信息显示完成，按下 Ctrl-C 回到命令提示界面。查看输出信息，我们会看到很多地方重复提到[sdb]，这正好与 SCSI 类型的硬盘设备名相匹配。了解了这一点，我们就自然会额外注意下面的两行信息：

```
Jul 23 10:07:59 linuxbox kernel:  sdb: sdb1
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Attached SCSI removable disk
```

以上信息告诉我们该设备名是/dev/sdb, /dev/sdb1 指的是此设备的第一个分区。正如大家所看到的，学习使用 Linux 是一项有趣的发现工作。

注意 tail -f /var/log/messages 命令行是用来进行实时系统监测的好方法。

获得设备名后，我们便可以挂载此闪存设备：

```
[me@linuxbox ~]$ sudo mkdir /mnt/flash
[me@linuxbox ~]$ sudo mount /dev/sdb1 /mnt/flash
[me@linuxbox ~]$ df
Filesystem          1K-blocks      Used Available Use% Mounted on
/dev/sda2            15115452   5186944   9775164  35% /
/dev/sda5            59631908  31777376  24776480  57% /home
/dev/sda1              147764     17277    122858  13% /boot
tmpfs                  776808         0    776808   0%  /dev/shm
/dev/sdb1               15560         0     15560   0%  /mnt/flash
```

只要设备一直与计算机保持连接并且系统没有重启，设备名就不会改变。

15.2 创建新的文件系统

将一个使用 FAT32 文件系统的闪存驱动器重新格式化为 Linux 本地文件系统，需要两个步骤：第一，（可选）在对现有的分区不满意的情况下创建一个新的分区布局；第二，在驱动器上创建一个新的空文件系统。

注意 下面是一个格式化闪存驱动器的例子。由于在使用过程中闪存中的所有内容都会被擦除，所以最好准备一个不含任何重要文件的驱动器。另外，要绝对确保指定的是自己系统上显示的实际设备名而不是直接用本文中使用的设备名，否则可能会格式化错误的驱动器！

15.2.1 用 fdisk 命令进行磁盘分区

fdisk 命令实现用户与磁盘设备（比如硬盘驱动器和闪存驱动器）进行较低

层次的直接交互。该工具可以用来编辑、删除以及创建设备分区。使用闪存前，我们必须首先将其卸载（如果需要的话）然后再启动 fdisk 程序：

```
[me@linuxbox ~]$ sudo umount /dev/sdb1
[me@linuxbox ~]$ sudo fdisk /dev/sdb
```

请注意使用 fdisk 命令指定设备时，设备名要是整个设备的而不是分区号。程序启动后，会出现下面的提示信息：

```
Command (m for help):
```

输入 m 后会出现下面的程序菜单：

```
Command action
   a   toggle a bootable flag
   b   edit bsd disklabel
   c   toggle the dos compatibility flag
   d   delete a partition
   l   list known partition types
   m   print this menu
   n   add a new partition
   o   create a new empty DOS partition table
   p   print the partition table
   q   quit without saving changes
   s   create a new empty Sun disklabel
   t   change a partition's system id
   u   change display/entry units
   v   verify the partition table
   w   write table to disk and exit
   x   extra functionality (experts only)

Command (m for help):
```

首先查看现有磁盘分区布局，可以通过输入字母 p 打印显示设备的分区表：

```
Command (m for help): p

Disk /dev/sdb: 16 MB, 16006656 bytes
1 heads, 31 sectors/track, 1008 cylinders
Units = cylinders of 31 * 512 = 15872 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1               2        1008      15608+    b  W95 FAT32
```

可以看到此例中使用的是一个 16MB 大小的、仅有一个分区的闪存，并且只使用了可用的 1008 个磁柱（Cylinder）中的 1006 个。该分区被识别为是 Windows 95 FAT32 类型的分区，虽然有些程序会利用此身份标识来限制对磁盘可进行的操作种类，但大多数时候改不改变该标识无关紧要。然而，作为示范，我们还是需要将其改为 Linux 类型的分区。为此，首先我们得知道 Linux 分区使

用哪个身份识别 ID 码。从上表可以看出，字母 b 表示 Windows 95 FAT32 类型的分区。于是我们就需要查看有效分区类型对照表，参照先前的程序菜单，我们可以看到下面的菜单选项：

```
l  list known partition types
```

按照提示输入 l，一张包含所有可能分区类型的对照表便显示出来。查表，可以看到现有的分区类型用 b 表示，而 Linux 分区类型则用 83 表示。

回到程序菜单，会看到用来改变分区 ID 的菜单选项：

```
t  change a partition's system id
```

在提示命令框中输入 t 和新的 ID：

```
Command (m for help): t
Selected partition 1
Hex code (type L to list codes): 83
Changed system type of partition 1 to 83 (Linux)
```

完成了分区 ID 的修改，到目前为止，设备一直处于未开发状态（所有的变化都储存在了内存中而非物理设备上），所以下一步我们就该向设备写入修改后的分区表，然后退出。

在提示界面中输入 w 完成上述操作：

```
Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.

WARNING: If you have created or modified any DOS 6.x
partitions, please see the fdisk manual page for additional
information.
Syncing disks.
[me@linuxbox ~]$
```

如果不打算对设备做任何改动，可以输入 q，这样就可以不保存改动而退出程序了。在此过程中，我们可以毫无顾忌地忽略那些冠冕堂皇的警告信息。

15.2.2 用 mkfs 命令创建新的文件系统

分区编辑已经完成（虽然可能无足轻重），我们便可以在闪存上创建新的文件系统。mkfs（make filesystem 的缩写）命令可以用来创建各种类型的文件系统，例如我们要在设备上创建 ext3 文件系统，那就在想要格式化分区的设备名前面使用-t 参数选项指明创建的文件系统是 ext3 类型。

```
[me@linuxbox ~]$ sudo mkfs -t ext3 /dev/sdb1
mke2fs 1.40.2 (12-Jul-2012)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
3904 inodes, 15608 blocks
780 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=15990784
2 block groups
8192 blocks per group, 8192 fragments per group
1952 inodes per group
Superblock backups stored on blocks:
        8193

Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 34 mounts or
180 days, whichever comes first. Use tune2fs -c or -i to override.
[me@linuxbox ~]$
```

当 ext3 是指定的文件系统类型，该程序会输出许多信息。如果要将该设备重新格式化为原来的 FAT32 文件系统，则用 vfat 作为-t 的选项参数。

```
[me@linuxbox ~]$ sudo mkfs -t vfat /dev/sdb1
```

这种分区及格式化过程适用于任何有额外存储设备插入系统的时候。虽然此处仅仅用闪存作为例子讲解，但这样的过程同样适用于内部硬盘以及其他像 USB 磁盘驱动器之类的可移动存储设备。

15.3 测试、修复文件系统

前面在讨论/etc/fstab 文件的时候，我们会看到每一行的末尾有许多神秘的数字。系统每次启动时，挂载文件系统前都会惯例性地检查文件系统的完整性，此检查过程是由 fsck（filesystem check 的缩写）程序完成的，fstab 文件每个条目末尾的数字正是对应设备的检查优先级。由上例中的 fstab 文件可知，根目录首先被检查，然后就是 home 目录和 boot 目录，末尾数字是 0 的设备表示不执行惯例性检查。

除了检查文件系统的完整性外，fsck 还能修复损坏的文件系统，修复程度取决于损坏程度。对于类 UNIX 文件系统，已修复的文件会存放在文件系统根目录下的 lost + found 目录中。

下面的命令可以用来检查闪存（闪存事先应该已卸载）

```
[me@linuxbox ~]$ sudo fsck /dev/sdb1
fsck 1.40.8 (13-Mar-2012)
e2fsck 1.40.8 (13-Mar-2012)
/dev/sdb1: clean, 11/3904 files, 1661/15608 blocks
```

依据我的经验，除非出现像磁盘驱动器不能工作这样的硬件问题，不然文件系统损坏是非常少见的。多数系统，如果在启动期间检测到文件系统损坏，系统会自动停止加载并且引导你运行 fsck 程序。

fsck 是什么

在 UNIX 文化中，fsck 通常用来取代一个与它有 3 个字母相同的流行词（fuck）。这确实很合适，因为当你处于被迫运行 fsck 程序的情况时，你很有可能不自然地就会蹦出 fuck。

15.4 格式化软盘

对于那些仍然使用配备软盘驱动器的老式电脑的用户，同样可以管理这些软盘设备。如何准备一张空白软盘？首先，我们对软盘进行一个低级格式化操作，然后创建一个文件系统。dformat 程序后面输入指定的软盘设备名（通常是/dev/fd0）即可完成格式化操作：

```
[me@linuxbox ~]$ sudo fdformat /dev/fd0
Double-sided, 80 tracks, 18 sec/track. Total capacity 1440 kB.
Formatting ... done
Verifying ... done
```

接下来用 mkfs 命令为软盘创建一个 FAT 文件系统。

```
[me@linuxbox ~]$ sudo mkfs -t msdos /dev/fd0
```

请注意，这里我们使用 msdos 文件系统类型以期获得更小更早类型的文件分配表。软盘准备工作完成后，就可以像其他设备一样被挂载。

15.5 直接从/向设备转移数据

虽然，我们通常认为电脑上的数据都是以文件的形式存储的，但也有可能会认为数据以“原始”形式存储。以磁盘驱动器为例，它包含了许多被操作系

统当作目录或文件的数据"块"。如果可以简单地把磁盘驱动器当作一个大数据块集，那么我们就可以执行一些有用任务，诸如克隆设备等。

dd 程序可以完成这样的任务，该命令将数据块从一个地方复制到另一个地方。由于历史原因，该命令使用句法比较独特，其格式如下：

```
dd if=input_file of=output_file [bs=block_size [count=blocks]]
```

假定现在有两个容量一样的 U 盘，并且我们希望将第一个 U 盘里面的内容准确完全地复制到第二个 U 盘里面。如果这两个盘都已经连接到电脑上并且它们的设备名分别为/dev/sdb 和设备/dev/sdc，那么我们就可以用下面的命令行将第一个盘上的所有内容复制到第二个盘上：

```
dd if=/dev/sdb of=/dev/sdc
```

或者，如果只有第一个盘连接到电脑上，那么我们可以将其内容先复制到一个普通的文件里以备后续储存或复制：

```
dd if=/dev/sdb of=flash_drive.img
```

警告 dd 命令功能很强大。尽管该命令是由 data definition 两个单词缩写而来，但有时亦被称为"destroy disk"（摧毁磁盘），因为使用者经常把 if 或是 or 后面指定的设备名输错。一定要记住命令输入结束按 Enter 键前要反复检查你指定的输入输出是否与你的本意一致。

15.6 创建 CD-ROM 映像

向 CD-ROM（CD-R 或是 CD-RW）写入数据包括两个步骤：首先，创建一个 ISO 映像文件，也就是 CD-ROM 文件系统映像；其次，将此映像文件写入到 CD-ROM（只读光盘）介质中。

15.6.1 创建一个 CD-ROM 文件映像副本

如果我们想给现有的 CD-ROM 创建一个 ISO 映像，可以使用 dd 命令将 CD-ROM 中所有数据块复制到本地某个文件。假定我们有一张 Ubuntu 的 CD 光盘，并打算创建一个 ISO 文件以便后续制作更多副本。CD 光盘被插入且其设备名被确定后（假设是/dev/cdrom），我们便可以制作 ISO 文件：

```
dd if=/dev/cdrom of=ubuntu.iso
```

该方法同样适用于数据类 DVD，但不适用于音频 DVD，因为音频 DVD 并不使用文件系统实现存储。对于音频 CD，可以使用 cdrdao 命令。

用其他名字命名的程序……

如果你曾经看过创建和刻录 CD-ROM 和 DVD 之类的光学介质的在线教程，你肯定常听到 mkisofs 和 cdrecord 这两个程序。这两个程序包含在叫做 cdtools 的非常受欢迎的软件包里，Jörg Schilling 是该软件包的作者。2006 年的夏天，Schilling 先生对 cdtools 软件包中的部分内容提出了许可证变更，但是这些变更在许多 Linux 社区的人看来与 GNU 通用公共许可证不兼容。结果，cdtools 项目便开始分叉了，包括 cdrecord 和 mkisofs 程序分别被 wodim 和 genisoimage 取代。

15.6.2 从文件集合中创建映像文件

enisoimage 程序通常用于创建包含一个目录内容的 ISO 映像文件。首先我们创建一个目录，该目录包含了所有我们希望加进映像文件里的文件，然后运行 enisoimage 程序创建映像文件。例如，假使事先我们已经创建了一个叫做 ~/cd-rom-files 的文件目录，并且该目录中包含了所有准备装入 CD-ROM 中的文件，那么接下来，我们就可以用 enisoimage 命令创建 ISO 映像文件，示例如下：

```
genisoimage -o cd-rom.iso -R -J ~/cd-rom-files
```

-R 选项添加了允许 Rock Ridge 延伸的元数据，此延伸允许在 Linux 中使用较长文件名的文件以及 POSIX 风格的文件。同样，-J 选项允许 Joliet 延伸，此延伸允许在 Windows 中使用较长文件名的文件。

15.7 向 CD-ROM 写入映像文件

映像文件创建好后，下一步便是将其刻录进光学介质中。下面我们所讨论的大部分命令都适用于 CD-ROM 和 DVD 介质。

15.7.1 直接挂载 ISO 映像文件

当 ISO 映像文件仍在硬盘上时，我们可以把它当作已存在于光学介质中，

并且有一个窍门可以实现该映像文件的挂载。那就是通过增加-o loop 选项来挂载（与-t iso9660 文件系统类型参数一起使用），如此便可以把映像文件当作设备一样挂载在文件系统树上了。

```
mkdir /mnt/iso_image
mount -t iso9660 -o loop image.iso /mnt/iso_image
```

在上面的例子中，我们创建了一个叫做/mnt/iso_image 挂载节点并将 image.iso 映像文件挂载在该节点上。映像文件挂载成功后，就可以把它当作真实的 CD-ROM 或 DVD。记住，当不需要该映像文件的时候要将其卸载。

15.7.2 擦除可读写 CD-ROM

可擦写 CD-ROM 在重用之前需要被擦除或清空，我们可以通过 wodim 命令指定光盘刻录机操作对象的设备名以及所要执行的擦除类型来完成。wodim 程序提供多种擦除类型，最基本的就是 fast 类型。

```
wodim dev=/dev/cdrw blank=fast
```

15.7.3 写入映像文件

同样我们使用 wodim 命令写入映像文件，通过指定写入的光介质刻录设备的名字以及映像文件的名字来完成：

```
wodim dev=/dev/cdrw image.iso
```

除了设备名和映像文件名，wodim 命令还支持很多的选项。两个比较常见的就是-v 和-dao 选项，-v 选项强调输出显示详细信息，而-dao 选项则决定光盘刻录采用一次刻录模式，这种模式一般用于光盘商业化生产。wodim 命令的默认模式是一次轨道模式，一般用于录制音乐曲目。

15.8 附加认证

通常，确认所下载的 ISO 映像文件是否完整大有必要。多数情况下，ISO 映像文件的发行商会提供一个校验和文件，校验和是通过一种奇异的数学计算得到而以数字形式表示的计算值，它代表了目标文件的内容。即便文件内容只有某个位发生了变化，校验和的结果也会有很大不同。通常我们使用 md5sum 程序生成校验和，示例如下，输出结果是一个独特的十六进制数：

```
md5sum image.iso
34e354760f9bb7fbf85c96f6a3f94ece  image.iso
```

映像文件下载结束后，你可以用 md5sum 命令得出其校验和，并与供应商提供的 md5sum 校验和数值进行比较验证。

md5sum 程序除了用来检查下载文件的完整性外，还可以用来验证新刻录的光学介质。检验光学介质，首先生成映像文件的校验和，然后再生成光学介质的校验和，比较这两个校验和，就可以判别映像文件是否完整地写入光学介质。该算法的关键在于如何只计算光学介质中包含该映像文件部分的校验和，常用的方法如下，计算映像文件中包含的 2K 字节块的数量（光学介质总是以 2K 字节的块为单位进行擦写），然后从光学介质中读取同样多数量的块，计算这些块的校验和。其实，有些类型的光学介质是不需要计算块数量的，像一次刻录模式刻录的 CD-R 就可以直接用下面的命令行检验：

```
md5sum /dev/cdrom
34e354760f9bb7fbf85c96f6a3f94ece  /dev/cdrom
```

但是也有许多类型的介质像 DVD 这类的，需要精确计算块的数量，示例如下。该命令行检查了 dvd-image.iso 映像文件的完整性以及 DVD 播放器内/dev/dvd 光盘的完整性。你能弄明白该命令行的工作原理吗？

```
md5sum dvd-image.iso; dd if=/dev/dvd bs=2048 count=$(( $(stat -c "%s" dvd-image
.iso) / 2048 )) | md5sum
```

第16章 网络

在网络连接方面，Linux 可以说是万能的。Linux 工具可以建立各种网络系统及应用，包括防火墙、路由器、域名服务器、NAS（网络附加存储）盒等。

由于网络连接涉及的领域很广，所以用于控制、配置网络的命令自然很多。本章只着重讲一些经常用到的命令，涉及网络监测以及文件传输等方面。此外，我们还会探讨一下用于远程登录的 ssh 命令，所涉及的命令如下所示。

- ping：向网络主机发送 ICMP　ECHO_REQUEST 数据包。
- traceroute：显示数据包到网络主机的路由路径。
- netstat：显示网络连接、路由表、网络接口数据、伪连接以及多点传送成员等信息。
- ftp：文件传输命令。
- lftp：改善后的文件传输命令。
- wget：非交互式网络下载器。

- ssh：OpenSSH（SSH 协议的免费开源实现）版的 SSH 客户端（远程系统登录命令）。
- scp：secure copy 的缩写，是远程复制文件命令。
- sftp：secure file transfer program 的缩写，安全文件传输程序。

接下来给大家补充一些网络方面的背景知识。在当今这个互联网时代，每一个计算机用户都需要对网络这个概念有一个基本了解。便于更好地理解本章内容，首先熟悉下面的术语。

- IP (Internet protocol) address：互联网协议地址。
- host and domain name：主机名和域名。
- URI (uniform resource identifier)：统一资源标识符。

注意　下面涉及到的一些命令可能需要安装额外的软件包（取决于使用的 Linux 版本），当然这些软件包可以从所使用的 Linux 版本的库源中下载。另外，有些命令可能要求超级用户的权利才能执行。

16.1 检查、监测网络

即使你不是系统管理员，经常检查网络的性能和运行情况也是有必要的。

16.1.1 ping——向网络主机发送特殊数据包

最基本的网络连接命令就是 ping 命令。ping 命令会向指定的网络主机发送特殊网络数据包 IMCP ECHO_REQUEST。多数网络设备收到该数据包后会做出回应，通过此法即可验证网络连接是否正常。

注意　有时从安全角度出发，通常会配置部分网络通信设备（包括 Linux 主机）以忽略这些数据包，因为这样可以降低主机遭受潜在攻击者攻击的可能性。当然，防火墙经常被设置为阻碍 IMCP 通信。

例如，想要验证是否可以登录网站 http://www.linuxcommand.org/（我最喜欢登录的网站之一），可以按如下方式使用 ping 命令。

```
[me@linuxbox ~]$ ping linuxcommand.org
```

一旦程序启动，ping 命令便以既定的时间间隔（默认是 1s）传送数据包直到该命令被中断。

```
[me@linuxbox ~]$ ping linuxcommand.org
PING linuxcommand.org (66.35.250.210) 56(84) bytes of data.
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=1 ttl=43 time=10
7 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=2 ttl=43 time=10
8 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=3 ttl=43 time=10
6 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=4 ttl=43 time=10
6 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=5 ttl=43 time=10
5 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=6 ttl=43 time=10
7 ms

--- linuxcommand.org ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 6010ms
rtt min/avg/max/mdev = 105.647/107.052/108.118/0.824 ms
```

按 Ctrl-C 键终止 ping 程序（此时正好是本例中第 6 个数据包传输结束后），ping 程序会将反映运行情况的数据显示出来。数据包丢失 0%表示网络运行正常，ping 连接成功则表明网络各组成成员（接口卡、电缆、路由和网关）总体处于良好的工作状态。

16.1.2 traceroute——跟踪网络数据包的传输路径

traceroute 程序（有些系统则使用功能相仿的 tracepath 程序代替）会显示文件通过网络从本地系统传输到指定主机过程中所有停靠点的列表。下例列出了数据在连接到网站 http://www.slashdot.org/时所经过的站点。

```
[me@linuxbox ~]$ traceroute slashdot.org
```

输出如下内容。

```
traceroute to slashdot.org (216.34.181.45), 30 hops max, 40 byte packets
 1  ipcop.localdomain (192.168.1.1)  1.066 ms  1.366 ms  1.720 ms
 2  * * *
 3  ge-4-13-ur01.rockville.md.bad.comcast.net (68.87.130.9)  14.622 ms  14.885
ms  15.169 ms
 4  po-30-ur02.rockville.md.bad.comcast.net (68.87.129.154)  17.634 ms  17.626
ms  17.899 ms
 5  po-60-ur03.rockville.md.bad.comcast.net (68.87.129.158)  15.992 ms  15.983
ms  16.256 ms
 6  po-30-ar01.howardcounty.md.bad.comcast.net (68.87.136.5)  22.835 ms  14.23
3 ms  14.405 ms
```

```
 7  po-10-ar02.whitemarsh.md.bad.comcast.net (68.87.129.34)  16.154 ms  13.600
ms  18.867 ms
 8  te-0-3-0-1-cr01.philadelphia.pa.ibone.comcast.net (68.86.90.77)  21.951 ms
21.073 ms  21.557 ms
 9  pos-0-8-0-0-cr01.newyork.ny.ibone.comcast.net (68.86.85.10)  22.917 ms  21
.884 ms  22.126 ms
10  204.70.144.1 (204.70.144.1)  43.110 ms  21.248 ms  21.264 ms
11  cr1-pos-0-7-3-1.newyork.savvis.net (204.70.195.93)  21.857 ms cr2-pos-0-0-
3-1.newyork.savvis.net (204.70.204.238)  19.556 ms cr1-pos-0-7-3-1.newyork.sav
vis.net (204.70.195.93)  19.634 ms
12  cr2-pos-0-7-3-0.chicago.savvis.net (204.70.192.109)  41.586 ms  42.843 ms
cr2-tengig-0-0-2-0.chicago.savvis.net (204.70.196.242)  43.115 ms
13  hr2-tengigabitethernet-12-1.elkgrovech3.savvis.net (204.70.195.122)  44.21
5 ms  41.833 ms  45.658 ms
14  csr1-ve241.elkgrovech3.savvis.net (216.64.194.42)  46.840 ms  43.372 ms  4
7.041 ms
15  64.27.160.194 (64.27.160.194)  56.137 ms  55.887 ms  52.810 ms
16  slashdot.org (216.34.181.45)  42.727 ms  42.016 ms  41.437 ms
```

由该列表可知，从测试系统到 http://www.slashdot.org/网站的连接需要经过16 个路由器。对于那些提供身份信息的路由器，此列表则列出了它们的主机名、IP 地址以及运行状态信息，这些信息包含了文件从本地系统到路由器 3 次往返时间。而对于那些因为路由器配置、网络堵塞或是防火墙等原因不提供身份信息的路由器，则直接用星号行表示，如上面输出信息中的 2 号停靠站。

16.1.3 netstat——检查网络设置及相关统计数据

netstat 程序可以用于查看不同的网络设置及数据。通过使用其丰富的参数选项，我们可以查看网络启动过程的许多特性。示例如下，使用-ie 选项，我们可以检查系统中的网络接口信息。

```
[me@linuxbox ~]$ netstat -ie
eth0      Link encap:Ethernet  HWaddr 00:1d:09:9b:99:67
          inet addr:192.168.1.2 Bcast:192.168.1.255 Mask:255.255.255.0
          inet6 addr: fe80::21d:9ff:fe9b:9967/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:238488 errors:0 dropped:0 overruns:0 frame:0
          TX packets:403217 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:153098921 (146.0 MB)  TX bytes:261035246 (248.9 MB)
          Memory:fdfc0000-fdfe0000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:2208 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2208 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:111490 (108.8 KB)  TX bytes:111490 (108.8 KB)
```

以上的输出信息显示，测试系统有两个网络端口：第一个称为 eth0，是以太网端口；第二个称为 lo，是系统用来自己访问自己的回环虚拟接口。

对网络进行日常诊断，关键是看能否在每个接口信息第四行的开头找到 UP 这个词以及能否在第二行的 inet addr 字段找到有效的 IP 地址。第四行的 UP 代表着该网络接口已启用，而对于使用动态主机配置协议的系统（DHCP），inet addr 字段里面的有效 IP 地址则说明了 DHCP 正在工作。

使用-r 选项将显示内核的网络路由表，此表显示了网络之间传送数据包时网络的配置情况。

```
[me@linuxbox ~]$ netstat -r
Kernel IP routing table
Destination Gateway      Genmask       Flags  MSS Window  irtt Iface
192.168.1.0 *           255.255.255.0 U        0 0         0 eth0  default
192.168.1.1 0.0.0.0       UG       0 0          0 eth0
```

此例显示的是运行在防火墙/路由器后面的局域网（LAN）上一客户端的典型路由表。该表的第一行表示接收方的 IP 地址为 192.168.1.0，IP 以 0 结尾表示接收方是网络而非个人主机，也就是说接收方可以是局域网（LAN）上的任何主机。下面的 Gateway 参数字段，表示的是建立当前主机与目标网络之间联系的网关（或路由）的名称或 IP 地址，此参数值是星号表示无需网关。

最后一行包含默认的接收方，这意味着所有通信都以该网络为目的地。上例中，网关被定义为 IP 地址是 192.168.1.1 的路由器，该路由器对双方通信做出最佳路径判断。

netstat 程序也有很多参数选项，而以上只列举了其中两个，要了解其整个参数列表可以查看 netstat 的 man 手册页。

16.2 通过网络传输文件

只有掌握了如何通过网络转移文件，才会明白网络的作用之大。有许多命令可以用于传送网络数据，现在我们就介绍其中的两个，后续章节将会介绍更多。

16.2.1 ftp——采用 FTP（文件传输协议）传输文件

ftp 是 Linux 比较经典的命令之一，由 File Transfer Protocol 协议缩写而来。ftp 用作下载文件工具在因特网上使用很广泛，大多数 Web 浏览器都支持 ftp 命令，读者肯定也经常遇到以 ftp://协议开头的 URI。

ftp 程序比 Web 浏览器出现得早，它用来与 FTP 服务器进行通信，所谓 FTP 服务器就是那些包含供网络上传、下载文件的机器。

FTP（原来的表示形式）并不安全，因为它以明文的方式传送账户名以及密码。这意味着这些信息并没有加密，任何一个接触网络的人都能看到它们。鉴于此，几乎所有使用 FTP 协议进行的网络文件传输都是由匿名 FTP 服务器处理的。匿名服务器允许任何人使用 anonymous 登录名以及无意义的密码登录。

下面是一个典型的 ftp 会话，其功能是从匿名 FTP 服务器 fileserver 上的 /pub/cd_images/Ubuntu-8.04 目录中下载 Ubuntu ISO 映像文件。

```
[me@linuxbox ~]$ ftp fileserver
Connected to fileserver.localdomain.
220 (vsFTPd 2.0.1)
Name (fileserver:me): anonymous
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd pub/cd_images/Ubuntu-8.04
250 Directory successfully changed.
ftp> ls
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
-rw-rw-r--    1 500     500     733079552 Apr 25 03:53 ubuntu-8.04-desktop-
i386.iso
226 Directory send OK.
ftp> lcd Desktop
Local directory now /home/me/Desktop
ftp> get ubuntu-8.04-desktop-i386.iso
local: ubuntu-8.04-desktop-i386.iso remote: ubuntu-8.04-desktop-i386.iso
200 PORT command successful. Consider using PASV.
150 Opening BINARY mode data connection for ubuntu-8.04-desktop-i386.iso
(733079552 bytes).
226 File send OK.
733079552 bytes received in 68.56 secs (10441.5 kB/s)
ftp> bye
```

表 16-1 列出了这段代码中所输入的每个命令行的含义与解释。

表 16-1 ftp 命令实例

命令	代表的含义
ftp fileserver	启动 ftp 程序，建立与 FTP 服务器 fileserver 的连接
anonymous	登录名，登录提示框出现之后就是密码输入提示框。一些服务器可以接受空白密码，其他的则要求密码以邮件地址的形式。在那样的情况下，就尝试 user@example.com 这样的格式
cd pub/cd_images/Ubuntu-8.04	打开远程系统上含有所需文件的目录。注意，对于多数匿名服务器，供公开下载的文件一般都存放在 pub 目录下

续表

命令	代表的含义
ls	列出远程系统上的目录列表
lcd Desktop	切换至本地系统的 ~/Desktop 目录。本例中，ftp 程序是在 home（~）目录下启动的，此命令行将工作目录切换至~/Desktop 下
get ubuntu-8.04-desktop-i386.iso	告诉远程系统将 ubuntu-8.04-desktop-i386.iso 映像文件发送给本地系统。由于本地系统的工作目录已经切换至~/Desktop 下，映像文件也会下载到该目录下
bye	注销登录远程服务器并且结束 ftp 程序。也可以使用 quit 或 exit 命令代替

在提示符 ftp>后面输入 help 会显示 ftp 所支持的命令列表。在已被授予足够权限的服务器上使用 ftp 命令，可以执行许多常见的文件管理任务。虽然这很笨拙，但这不失为一种办法。

16.2.2 lftp——更好的 ftp（文件传输协议）

ftp 并不是唯一的命令行 FTP 客户端。事实上，有很多这样的命令行。其中更好用也更受欢迎的一个就是由 Alexander Lukyanov 编写的 lftp 命令，它与传统的 ftp 程序功能类似但却有很多额外的便利功能，包括多协议支持（HTTP）、下载失败时自动重新尝试、后台进程支持、Tab 键完成文件名输入等许多其他的功能。

16.2.3 wget——非交互式网络下载工具

wget 是另一个用于文件下载的命令行程序。该命令既可以用于从网站上下载内容也可以用于从 FTP 站点下载，单个文件、多个文件甚至整个网站都可以被下载。下例演示的就是用 wget 命令下载网站 http://www.linuxcommand.org/第一页内容。

```
[me@linuxbox ~]$ wget http://linuxcommand.org/index.php
--11:02:51--  http://linuxcommand.org/index.php
           => `index.php'
Resolving linuxcommand.org... 66.35.250.210
Connecting to linuxcommand.org|66.35.250.210|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]

    [ <=>                                 ] 3,120        --.--K/s
11:02:51 (161.75 MB/s) - `index.php' saved [3120]
```

wget 命令的许多参数选项支持递归下载、后台文件下载（允许下线的情况下继续下载）以及继续下载部分被下载的文件等操作。这些特点都清楚的写在该命令的 better-than-average（优于平均水平）man 手册页中。

16.3 与远程主机的安全通信

多年以前，类 UNIX 操作系统就可以通过网络进行远程操控。早期，在互联网还未普及的时候，登录远程主机有两个很受欢迎的命令——rlogin 和 telnet 命令。但是它们与 ftp 命令有着相同的致命缺点，即所有通信信息（包括用户名和密码）都是以明文的方式传输的，所以它们并不适用于互联网时代。

16.3.1 ssh——安全登录远程计算机

为了解决明文传送的问题，一个叫做 SSH（Secure Shell 的缩写）的新协议应运而生。SSH 协议解决了与远程主机进行安全通信的两个基本问题：第一，该协议能验证远程主机的身份是否真实，从而避免中间人攻击；第二，该协议将本机与远程主机之间的通信内容全部加密。

SSH 协议包括两个部分：一个是运行在远程主机上的 SSH 服务端，用来监听端口 22 上可能过来的连接请求；另一个是本地系统上的 SSH 客户端，用来与远程服务器进行通信。

多数 Linux 发行版都采用 BSD 项目的 openSSH（SSH 的免费开源实现）方法实现 SSH。有些发行版如 Red Hat 会默认包含客户端包和服务端包，而有的版本如 Ubuntu 则仅仅提供客户端包。系统想要接收远程连接，就必须安装、配置以及运行 OpenSSH-server 软件包，并且必须允许 TCP 端口 22 上进来的网络连接（当服务器正在运行防火墙或是在防火墙后面时）。

注意 如果没有可以连接的远程系统但却要尝试本例，可以在系统已安装了 OpenSSH-server 软件包的基础上将远程主机名设为本地主机名。这样，机器便会与自身建立网络连接。

---

ssh 命令作为 SSH 客户端程序用于建立与远程 SSH 服务器之间的通信再合适不过了。如下便是使用 ssh 客户端程序来建立与远程主机 remote-sys 的连接的例子。

```
[me@linuxbox ~]$ ssh remote-sys
The authenticity of host 'remote-sys (192.168.1.4)' can't be established.
RSA key fingerprint is 41:ed:7a:df:23:19:bf:3c:a5:17:bc:61:b3:7f:d9:bb.
Are you sure you want to continue connecting (yes/no)?
```

第一次尝试连接的时候，由于 ssh 程序从来没有接触过此远程主机，所以会跳出一条“不能确定远程主机真实性”的消息。当出现这条警告消息的时候输入 yes 接受远程主机的身份，一旦建立了连接，会提示用户输入密码。

```
Warning: Permanently added 'remote-sys,192.168.1.4' (RSA) to the list of known hosts.
me@remote-sys's password:
```

密码输入正确后，远程系统的 shell 提示符便出现了。

```
Last login: Tue Aug 30 13:00:48 2011
[me@remote-sys ~]$
```

远程 shell 对话将一直开启着，直到用户在该对话框中输入 exit 命令断开与远程系统的连接。连接一旦断开后，本地 shell 会话恢复，本地 shell 提示符又重新出现。

使用非本地系统上使用的用户名也可以登录远程系统。例如，当本地用户 me 在远程系统上有一个 bob 账户，me 用户就可以用下面的命令登录远程系统上的 bob 账户。

```
[me@linuxbox ~]$ ssh bob@remote-sys
bob@remote-sys's password:
Last login: Tue Aug 30 13:03:21 2011
[bob@remote-sys ~]$
```

正如前面所说，ssh 命令会验证远程主机的真实性。如果远程主机没有成功验证，就会跳出下面的警告信息。

```
[me@linuxbox ~]$ ssh remote-sys
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
41:ed:7a:df:23:19:bf:3c:a5:17:bc:61:b3:7f:d9:bb.
Please contact your system administrator.
Add correct host key in /home/me/.ssh/known_hosts to get rid of this message.
Offending key in /home/me/.ssh/known_hosts:1
RSA host key for remote-sys has changed and you have requested strict
checking.
Host key verification failed.
```

一般有两个原因导致此警告信息：第一，有攻击者正在尝试中间人攻击，

这种情况很少见，因为大家都知道 ssh 会提醒用户发生了这样的攻击；第二，也是更有可能的原因便是远程系统在某种程度上改变了，例如，远程主机重新安装了操作系统或者 SSH 服务端。然而，从安全性上考虑，不应该因为发生第一种情况的可能性小就直接忽略，该警告出现时仍然要向远程系统的管理者核对。

确定该警告是由良性原因导致后，在客户端修正问题就安全了，解决方法就是用某种文本编辑器（也许是 vim）从~/.ssh/known_hosts 文件中移除过时的密钥。警告当中包含如下这句话。

```
Offending key in /home/me/.ssh/known_hosts:1
```

这意味着 known_hosts 文件的第一行包含了相悖的密钥。将该行删除后，ssh 程序就能重新获取远程系统新的身份验证凭据了。

ssh 命令除了能开启远程系统上的 shell 会话外，还能在远程系统上执行单个简单命令。例如，我们可以在 remote-sys 远程主机上执行 free 命令并将其结果直接输出到本地系统上。

```
[me@linuxbox ~]$ ssh remote-sys free
me@twin4's password:
             total      used      free    shared   buffers    cached
Mem:        775536    507184    268352         0    110068    154596
-/+ buffers/cache:    242520    533016
Swap:      1572856         0   1572856
[me@linuxbox ~]$
```

该特性可以有更有趣的用途，比如在远程系统上执行 ls 命令后直接将运行结果输出到本地系统的文件中。

```
[me@linuxbox ~]$ ssh remote-sys 'ls *' > dirlist.txt
me@twin4's password:
[me@linuxbox ~]$
```

请注意命令行中的单引号，之所以使用单引号是因为我们并不希望本该在远程主机上进行的路径扩展在本地系统上进行。同样，如果我们还希望执行结果能直接输出到远程系统的文件中，就应该将重定向符号和文件名一起置于单引号中。

```
[me@linuxbox ~]$ ssh remote-sys 'ls * > dirlist.txt'
```

SSH 的隧道技术

通过 SSH 与远程主机建立连接后，一个本地与远程系统之间的加密隧道就被建立了。通常，该隧道用于将在本地系统输入的命令安全地传送给远程

系统并将结果安全地传送回来。除了这样的基本功能外，SSH 协议还可以在本地与远程系统之间建立某种虚拟专用网络（VPN，Virtual Private Network），从而实现多种网络通信经此加密隧道传送。

也许，这一特性最常见的用途就是允许 X Window 系统之间的通信。在一个运行 X Window 服务器（也就是使用图形用户接口的机器）的系统上，可以远程启动和运行远程主机上的 X Window（图形化应用）客户端程序并将其图像化效果显示在本地系统上。这个过程操作起来很容易，假定我们正在操作一台运行 X Window 服务器的叫做 linuxbox 的本地主机，并打算在 remote-sys 远程主机上远程运行 xload 程序，还要在本地主机上看到该程序运行后的图形化效果，示例如下。

```
[me@linuxbox ~]$ ssh -X remote-sys
me@remote-sys's password:
Last login: Mon Sep 05 13:23:11 2011
[me@remote-sys ~]$ xload
```

xload 命令在远程系统上运行后，其图形窗口就会出现在本地系统上。然而对于某些系统，可能需要用-Y 参数选项而不是像上面所用的-X 选项完成该操作。

16.3.2 scp 和 sftp——安全传输文件

OpenSSH 软件包包含了两个使用 SSH 加密隧道进行网络间文件复制的程序， scp（secure copy 的缩写）便是其中之一。该命令与普通的文件复制命令 cp 类似，而它们之间最大的差别在于 scp 命令的源或目的地路径前面多个远程主机名和冒号。下面的例子实现了从 remote-sye 远程系统的 home 目录中将 document.txt 的文件复制到本地系统当前工作目录下的操作。

```
[me@linuxbox ~]$ scp remote-sys:document.txt .
me@remote-sys's password:
document.txt                          100% 5581    5.5KB/s   00:00
[me@linuxbox ~]$
```

与 ssh 命令一样，如果不是用本地系统的用户名登录远程系统，那么就需在远程主机名前添加将要登录的远程系统的账户名，示例如下。

```
[me@linuxbox ~]$ scp bob@remote-sys:document.txt .
```

另外一个 SSH 文件复制程序是 sftp。顾名思义，它是 ftp 程序的安全版本。sftp 与我们先前使用的 ftp 程序功能极为相似，只是 sftp 是用 SSH 加密隧道传输

信息而不是以明文方式传输。sftp 相比传统的 ftp 而言，还有一个重要的优点，就是它并不需要远程主机上运行 FTP 服务器，仅仅需要 SSH 服务器。这就意味着任何与 SSH 客户端连接的远程机器都可以当作 FTP 服务器使用，下面就是一个简单的会话实例。

```
[me@linuxbox ~]$ sftp remote-sys
Connecting to remote-sys...
me@remote-sys's password:
sftp> ls
ubuntu-8.04-desktop-i386.iso
sftp> lcd Desktop
sftp> get ubuntu-8.04-desktop-i386.iso
Fetching /home/me/ubuntu-8.04-desktop-i386.iso to ubuntu-8.04-desktop-i386.iso

/home/me/ubuntu-8.04-desktop-i386.iso 100% 699MB  7.4MB/s  01:35
sftp> bye
```

注意 Linux 发行版中许多图形文件管理器都支持 SFTP 协议。不管系统使用的是 Nautilus（GNOME）图形界面还是 Konqueror（KDE）图形界面，都可以在地址栏中输入以 sftp://开头的 URI，并对存储在运行 SSH 服务器的远程系统上的文件进行操作。

Windows 的 SSH 客户端

假使你正在操作一台 Windows 系统的计算机，但需要登录 Linux 服务器进行一些实际的操作。该怎么办？当然是为你的 Windows 计算机配置一个 SSH 客户端程序！有很多这样的工具。最受欢迎的可能要算 Simon Tatham 及其团队开发的 PuTTY 程序了。PuTTY 程序会显示一个终端窗口，并允许 Windows 用户在远程主机中打开一个 SSH（或 Telnet）会话，该程序也提供与 scp 和 sftp 命令功能类似的命令。

PuTTY 程序可以从 http://www.chiark.greenend.org.uk/~sgtatham/putty/网站上下载。

第 17 章 文件搜索

已经学习 Linux 这么长时间了，相信大家有一点已经很清楚，就是 Linux 系统含有非常多的文件！这就自然产生一个问题，“我们应该怎样查找东西？”。虽然我们已经知道，Linux 文件系统良好的组织结构，源自于类 UNIX 的操作系统代代传承的习俗，但是仅文件数量就会引起可怕的问题。

本章我们主要介绍两个用于在 Linux 系统中搜索文件的工具。

- locate：通过文件名查找文件。
- find：在文件系统目录框架中查找文件。

同时，我们也会介绍一个通常与文件搜索命令一起使用、处理搜索结果文件列表的命令。

- xargs：从标准输入中建立、执行命令行。

此外，我们还介绍了两个辅助工具。

- touch：更改文件的日期时间。

- stat：显示文件或文件系统的状态。

17.1 locate——较简单的方式查找文件

locate 命令通过快速搜索数据库，以寻找路径名与给定子字符串相匹配的文件，同时输出所有匹配结果。例如，假定查找名称以 zip 字符串开头的程序，由于查找的是程序文件，所以可以认为包含所要查找的程序的目录名应以 bin/结尾。因此，可以尝试下面的命令行。

```
[me@linuxbox ~]$ locate bin/zip
```

locate 程序将搜索该路径名数据库，并输出文件名包含字符串 bin/zip 的所有文件。

```
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

有时搜索需求并不是这么简单，这时便可以用 locate 命令结合其他诸如 grep 这样的工具实现一些更有趣的搜索。

```
[me@linuxbox ~]$ locate zip | grep bin
/bin/bunzip2
/bin/bzip2
/bin/bzip2recover
/bin/gunzip
/bin/gzip
/usr/bin/funzip
/usr/bin/gpg-zip
/usr/bin/preunzip
/usr/bin/prezip
/usr/bin/prezip-bin
/usr/bin/unzip
/usr/bin/unzipsfx
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

locate 程序已经使用了很长时间，因此出现了多种衍生体。slocate 和 mlocate 是现代 Linux 发行版本中最常见的两个衍生体，而它们通常都是由名为 locate 的符号链接访问。不同版本的 locate 有一些相同的选项设置，而有些版本则包括

正则表达式匹配（将会在第 19 章涉及）和通配符支持等。我们可以查看 locate 的 man 手册页确定系统安装的是哪个版本的 locate。

> **locate 的搜索数据库从何而来**
>
> 你也许曾注意过，有些 Linux 版本，系统刚刚安装好时 locate 命令并不能工作，但是如果第二天再尝试，就会发现它又能正常工作，这到底是怎么回事呢？其实，是因为 locate 的搜索数据库由另外一个叫做 updatedb 的程序创建，通常该程序作为一个 cron 任务定期执行。所谓 cron 任务就是指定期由 cron 守护进程执行的任务，多数装有 locate 命令的系统每天执行一次 updatedb 命令。由此可见，locate 的搜索数据库并不是持续更新的，所以 locate 命令查找不到非常新的文件。解决方法就是切换为超级用户，在提示框下手动运行 updatedb 程序。

17.2 find——较复杂的方式查找文件

locate 程序查找文件仅仅是依据文件名，而 find 程序则是依据文件的各种属性在既定的目录（及其子目录）里查找。本章将会花大量的篇幅介绍 find 命令的用法，因为它有很多有趣的特性会在后面有关编程概念的章节中多次讲到。

find 最简单的用法就是用户给定一个或是更多目录名作为其搜索范围。下面就用 find 命令列出当前系统主目录（~）下的文件列表清单。

```
[me@linuxbox ~]$ find ~
```

对于一些比较活跃的用户，一般系统内文件会比较多，使得上述命令行输出的列表肯定很长。不过，列表信息是以标准形式输出的，所以可以直接将此输出结果作为其他程序的输入。如下就是用 wc 程序计算 find 命令搜索到的文件的总量。

```
[me@linuxbox ~]$ find ~ | wc -l
47068
```

大家忙碌起来吧！find 命令的美妙之处就是可以用来搜索符合特定要求的文件，它通过综合应用 test 选项、action 选项以及 options 选项（看起来有点奇怪）实现高级文件搜索。下面让我们首先了解 test 选项。

17.2.1 test 选项

假定我们想要查找的是目录文件，我们可以添加下面的 test 选项达到此目的。

```
[me@linuxbox ~]$ find ~ -type d | wc -l
1695
```

添加 test 参数-type d 可以将搜索范围限制为目录，而下面例子中使用-type f 则表示只对普通文件进行搜索。

```
[me@linuxbox ~]$ find ~ -type f | wc -l
38737
```

表 17-1 列出了 find 命令支持的常用文件类型。

表 17-1 find 支持搜索的文件类型

文件类型	描述
b	块设备文件
c	字符设备文件
d	目录
f	普通文件
l	符号链接

另外我们还可以通过添加其他的 test 项参数实现依据文件大小和文件名的搜索。如下命令行就是用来查找所有符合*.JPG 通配符格式以及大小超过 1MB 的普通文件。

```
[me@linuxbox ~]$ find ~ -type f -name "*.JPG" -size +1M | wc -l
840
```

本例中添加的-name "*.JPG"的 test 选项表示查找的是符合.JPG 通配符格式的文件。注意，这里将通配符扩在双引号中是为了避免 shell 路径名扩展。另外添加的-size +1M test 选项，前面的加号表示查找的文件大小比给定的数值 1M 大。若字符串前面是减号则代表要比给定数值小，没有符号则表示与给定值完全相等。末尾的 M 是计量单位 MB（兆字节）的简写，表 17-2 中列出了每个字母与特定计量单位之间的对应关系。

表 17-2 find 支持的计量单位

字母	单位
b	512 字节的块（没有具体说明时的默认值）
c	字节
w	两个字节的字

续表

字母	单位
k	KB（每单位包含 1,024 字节）
M	MB（每单位包含 1,048,576 字节）
G	GB（每单位包含 1,073,741,824 字节）

find 命令支持多种 test 参数，表 17-3 概括了一些常见的参数。注意，前面所讲述的“+”和“-”号的用法适用于所有用到数值参数的情况。

表 17-3 find 命令的 test 项参数

test 参数	描述
-cmin n	匹配 n 分钟前改变状态（内容或属性）的文件或目录。如果不到 n 分钟，就用-n，如果超过 n 分钟，就用+n
-cnewer file	匹配内容或属性的修改时间比文件 file 更晚的文件或目录
-ctime n	匹配系统中 n*24 小时前文件状态被改变（内容、属性、访问权限等）的文件或目录
-empty	匹配空文件及空目录
-group name	匹配属于 name 组的文件或目录。name 可以描述为组名，也可以描述为该组的 ID 号
-iname pattern	与-name test 项功能类似只是不区分大小写
-inum n	匹配索引节点是 n 的文件。该 test 选项有助于查找某个特定索引节点上的所有硬件连接
-mmin n	匹配 n 分钟前内容被修改的文件或目录
-mtime n	匹配 n*24 小时前只有内容被更改的文件或目录
-name pattern	匹配有特定通配符模式的文件或目录
-newer file	匹配内容的修改时间比 file 文件更近的文件或目录。这在编写 shell 脚本进行文件备份的时候非常有用。每次创建备份时，更新某个文件（比如日志），然后用 find+此参数选项来确定上一次更新后哪个文件改变了
-nouser	匹配不属于有效用户的文件或目录。该 test 可以用来查找那些属于已删除账户的文件，也可以用来检测攻击者的活动
-nogroup	匹配不属于有效组的文件或目录
-perm mode	寻找访问权限与既定模式匹配的文件或目录。既定模式可以以八进制或符号的形式表示
-samefile name	与-inum test 选项类似。匹配与 file 文件用相同的 inode 号的文件
-size n	匹配 n 大小的文件
-type c	匹配 c 类型的文件
-user name	匹配属于 name 用户的文件和目录。name 可以描述为用户名也可以描述为该组的 ID 号

以上并不是 test 参数的完整列表，要了解其他相关内容可以查看 find 的 man 手册页。

操作符

即使拥有了 find 命令提供的所有 test 参数，我们仍然会需要一个更好的工具来描述 test 参数之间的逻辑关系。例如，如果我们需要确定某目录下是否所有的文件和子目录都有安全的访问权限，该怎么办？原则上就是去查找那些访问权限不是 0600 的文件和访问权限不是 0700 的子目录。幸运的是，find 命令的 test 选项可以结合逻辑操作从而建立具有复杂逻辑关系的匹配条件。我们可以用下面的命令行来满足上述 find 命令的匹配搜索。

```
[me@linuxbox ~]$ find ~ \( -type f -not -perm 0600 \) -or \( -type d -not -perm 0700 \)
```

这些都是什么？看起来太别扭了！不过，一旦你开始了解逻辑操作符，就会发现它们并没有那么复杂了（见表 17-4)。

表 17-4　find 命令的逻辑操作符

操作符	功能描述
-and（与操作）	查找使该操作符两边的检验条件都是真的匹配文件。有时直接缩写成-a。注意如果两个检测条件之间没有显式的显示操作符，and 就是默认的逻辑关系
-or（或操作）	查找使该操作符任何一边的检测条件为真的匹配文件。有时直接缩写成-o
-not（-非操作）	查找使该操作符后面的检测条件为假的匹配文件。有时直接缩写成-!
()（括号操作）	多个检测条件和逻辑操作符一起组成更长的表达式，而()操作就是用来区分逻辑表达式优先权的。默认的情况下，find 命令从左向右运算逻辑值。然而有时为了获得想要的结果必须扰乱默认的执行顺序，即便不需要，将一串字符表达式括起来对提高命令的可读性也很有帮助。请注意，括号字符在 shell 环境中有特殊意义，所以必须将它们在命令行中用引号引起来，这样才能作为 find 的参数传递。通常用反斜杠来避免这样的问题

对照这张逻辑操作符的列表，重新剖析上述的 find 命令行。从最全局的角度看，所有检测条件由一个或（or）操作符分成了两组。

（*表达式 1*） -or（*表达式 2*）

这样做是有原因的，因为我们查找的是具有某种权限设置的文件和另外一种权限设置的目录。那既然要同时查找文件和目录，怎么不是用 and 而是用 or？因为 find 命令在浏览扫描所有的文件和目录时，会判断每一个文件或目录是否匹配该 test 项检测条件。而我们的目标是具有不安全访问权限的文件或是具有不安全访问权限的目录，都知道匹配者不可能既是文件又是目录，所以不能用

and 逻辑关系。由此，可将表达式扩充如下。

```
(file with bad perms) -or (directory with bad perms)
```

接下来的问题就是如何判断文件或是目录具有“危险”权限，我们该怎么做呢？事实上，我们并不需要直接去寻找“危险”权限的文件或是目录，而是查找那些具有“‘不’好”权限的文件或目录，因为我们知道什么是“好的权限”。对于文件来说，权限是 0600 表示“好”（安全）的，而对于目录，“好”的权限应该是 0700。于是，判断具有“不好”访问权限的文件的表达式如下。

```
-type f -and -not -perms 0600
```

同样，判断具有“不好”访问权限的目录的表达式如下。

```
-type d -and -not -perms 0700
```

正如表 17-4 中提到的，and 操作是默认的，所以可以安全地移除。把如上表达式都整理到一起以下。

```
find ~ (-type f -not -perms 0600) -or (-type d -not -perms 0700)
```

然而，由于括号在 shell 环境下有特殊含义，所以我们必须对它们进行转义以防 shell 试图编译它们。在每个括号前加上反斜杠便可解决此问题。

逻辑运算符的另外一个特性也很值得大家了解，有如下两个被逻辑操作符分开的表达式。

```
expr1 -operator expr2
```

在任何情况下，表达式 expr1 都会被执行，而中间的操作符将决定表达式 expr2 是否被执行。表 17-5 列出了具体执行情况。

表 17-5　find 命令的 and/or 逻辑运算

表达式 expr1 的结果	逻辑操作	表达式 expr2 执行情况…
真	and	总是执行
假	and	不执行
真	or	不执行
假	or	总是执行

为什么会出现这样的情况？主要还是为了提高效率，以-and 逻辑运算为例，很明显表达式 expr1 -and expr2 的值在 expr1 为假的情况下不可能为真，所以就没有必要再去运算表达式 expr2 了。同样，对于表达式 expr1 -or expr2，在表达式 expr1 为真的情况下其逻辑值显然为真，于是就没有必要再运算表达式 expr2 了。

由此达到效率提高的目的。但为什么这一性质这么重要？这是因为这将直接影响下一步 actions 选项的动作行为，下面就来介绍 actions 选项。

17.2.2 action 选项

行动起来吧！前面 find 命令已经查找到所需要的文件，但是我们真正想做的是处理这些已查找到的文件。幸运的是，find 命令允许直接对搜索结果执行动作。

预定义动作

对搜索到的文件进行操作，即可以用诸多现成的预定义动作指令，也可以使用用户自定义的动作。首先来看一些预定义动作，见表 17-6。

表 17-6 预定义的 find 命令操作

动作	功能描述
-delete	删除匹配文件
-ls	对匹配文件执行 ls 操作，以标准格式输出其文件名以及所要求的其他信息
-print	将匹配的文件的全路径以标准形式输出。当没有指定任何具体操作时，该操作是默认操作
-quit	一旦匹配成功便退出

与 test 参数选项相比，actions 参数选项数量更多，可以参考 find 的 man 手册页获取更全面信息。

本章开头所举的第一个例子如下。

```
find ~
```

此命令行产生了一个包含当前系统主（~）目录中所有文件和子目录的列表。该列表之所以会在屏幕上显示出来，是因为在没有指定其他操作的情况下，-print 操作是默认的。因此，上述命令行等效于如下形式的命令行。

```
find ~ -print
```

当然也可以使用 find 命令删除满足特定条件的文件。示例如下，此命令行用于删除.BAK（这种文件一般是用来指定备份文件的）后缀的文件。

```
find ~ -type f -name '*.BAK' -delete
```

本例中，用户主目录及其子目录下的每个文件都被搜索了一遍匹配文件名以.BAK 结尾的文件。一旦被找到，则直接删除。

注意　毫无疑问，在使用-delete 操作时你一定要格外小心。最好先用-print 操作确认搜索结果后再执行-delete 删除命令。

在我们继续讲下一个知识点前，先来补充介绍一下逻辑运算是如何影响 find 的 action 操作的。仔细揣摩下面的命令。

```
find ~ -type f -name '*.BAK' -print
```

正如我们所知道的，该命令行用来查找所有文件名以.BAK 结尾的普通文件（-type f）并且以标准形式（-print）输出每个匹配文件的相关路径名。然而，该命令行之所以照这样的方式执行是由每个 test 选项和 action 选项之间的逻辑关系决定的。记住，每个 test 选项和 action 选项之间默认的逻辑关系是与（and）逻辑。下面的命令行逻辑关系能看得更清楚些。

```
find ~ -type f -and -name '*.BAK' -and -print
```

如上便是完整的表达式，而表 17-7 便列出了逻辑运算符是如何影响 action 操作的。

表 17-7　逻辑运算符的影响

检测条件/执行操作	在该情况下执行操作
-print	-type f 和-name '*.BAK'条件都匹配时，也就是文件是普通文件并且文件名也是以.BAK 结尾时
-name '*.BAK'	-type f 项匹配，也就是说文件是普通文件时
-type f	总是执行，因为是与逻辑关系中的第一个操作数

test 选项与 action 选项之间的逻辑关系决定了它们的执行情况，所以 test 选项和 action 选项的顺序很重要。例如，如果重新排列这些 test 选项和 action 选项，并将-print 操作作为逻辑运算的第一个操作数，那么命令行的运行结果将会有很大的不同。

```
find ~ -print -and -type f -and -name '*.BAK'
```

此命令行会把每个文件显示出来（因为-print 操作运算值总是为真），然后再对文件类型以及特定的文件扩展名进行匹配检查。

用户自定义操作

除了已有的预定义操作命令，同样也可以任意调用用户想要执行的操作命令。传统的方法就是像以下命令行使用-exec 操作。

```
-exec command {} ;
```

该格式中的 *command* 表示要执行的操作命令名，{}花括号代表的是当前路径，而分号作为必需的分隔符表示命令结束。使用-exec 完成-delete 操作示例如下。

```
-exec rm '{}' ';'
```

同样，由于括号和分号字符在 shell 环境下有特殊含义，所以在输入命令行时，要将它们用引号引起来或者用转义符隔开。

当然，交互式地执行用户自定义操作也不是不可能。通过使用-ok 操作取代原来的-exec 操作，每一次指定命令执行之前系统都会询问用户。

```
find ~ -type f -name 'foo*' -ok ls -l '{}' ';'
< ls ... /home/me/bin/foo > ? y
-rwxr-xr-x 1 me   me 224 2011-10-29 18:44 /home/me/bin/foo
< ls ... /home/me/foo.txt > ? y
-rw-r--r-- 1 me   me   0 2012-09-19 12:53 /home/me/foo.txt
```

上例中，查找文件名以 foo 字符串开始的文件，并且每次找到匹配文件后执行 ls –l 命令。-ok 操作会在 ls 命令执行之前询问用户是否执行。

提高效率

当使用-exec 操作时，每次查找到匹配文件后都会调用执行一次指定命令。但有时用户更希望只调用一次命令就完成对所有匹配文件的操作。例如，多数人可能更喜欢这样的命令执行方式。

```
ls -l file1
ls -l file2
```

而不是以下这样的方式。

```
ls -l file1 file2
```

第一种方式只需要执行命令一次而第二种方式则要多次重复执行。实现这样的一次操作有两种方法：一种方式比较传统，使用外部命令 xargs；另一种则是使用 find 本身自带的新特性。首先介绍下第二种方法。

通过将命令行末尾的分号改为加号，便可将 find 命令所搜索到的匹配结果作为指定命令的输入，从而一次完成对所有文件的操作。回到刚才的例子。

```
find ~ -type f -name 'foo*' -exec ls -l '{}' ';'
-rwxr-xr-x 1 me   me 224 2011-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me   me   0 2012-09-19 12:53 /home/me/foo.txt
```

每次找到匹配文件后就执行一次 ls 命令。将上述命令行改成下面的命令行。

```
find ~ -type f -name 'foo*' -exec ls -l '{}' +
-rwxr-xr-x 1 me   me 224 2011-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me   me   0 2012-09-19 12:53 /home/me/foo.txt
```

我们也能得到相同的结果，但是系统整体只执行一次 ls 命令。

同样我们可以使用 xargs 命令获得相同的效果，xargs 处理标准输入信息并将其转变为某指定命令的输入参数列表。结合前面的实例，我们可以这样使用 xargs 命令。

```
find ~ -type f -name 'foo*' -print | xargs ls -l
-rwxr-xr-x 1 me   me 224 2011-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me   me   0 2012-09-19 12:53 /home/me/foo.txt
```

该命令行中，find 命令的执行结果直接作为 xargs 输入，xargs 反过来将其转换成了 ls 命令的输入参数列表，最后执行 ls 操作。

注意 虽然一个命令行中可允许输入的参数有很多，但这并不表示可以无限输入，也存在命令行过长而使得 shell 编辑器无法承受的情况。如果命令行中包含的输入参数太多而超过了系统支持的最大长度，xargs 只会尽可能对最大数量的参数执行指定操作，并不断重复这一过程直到所有标准输入全部处理完毕。在 xargs 命令后面添加--show-limits 选项，即可知道命令行最大能承受的参数数量。

处理有趣的文件名

类 UNIX 系统允许文件名里面有嵌入的空格（甚至换行符！），这会给像 xargs 这些为其他程序创建参数列表的命令带来一些问题。因为内嵌的空格可能会被当做分隔符，而要执行的操作命令可能会把空格隔开的单词当做不同的输入参数来处理。为了解决这一问题，find 和 xargs 命令允许使用空字符作为参数之间的分隔符。在 ASCII 码中，空字符是由数字 0 代表的字符表示（而空格符在 ASCII 码中是由数字 32 代替）。find 命令提供-print0 这一 action 选项来产生以空字符作为各参数之间分隔符的输出结果，而 xargs 命令则有-null 参数选项支持 xargs 接收以空字符为参数分隔符的输入。示例如下。

```
find ~ -iname '*.jpg' -print0 | xargs --null ls -l
```

使用这一特性，可以确保所有的文件包括那些文件名中包含内嵌空格的文件也能得到正确的处理。

17.2.3 返回到 playground 文件夹

现在可以实际应用 find 命令了。首先让我们创建一个包含很多子目录及文件的 playground 文件夹平台。

```
[me@linuxbox ~]$ mkdir -p playground/dir-{00{1..9},0{10..99},100}
[me@linuxbox ~]$ touch playground/dir-{00{1..9},0{10..99},100}/file-{A..Z}
```

不得不惊叹于 Linux 命令行的威力！简单的两行命令，就创建了一个包含 100 个子目录的 playground 文件夹，并且每个子目录中又包含 26 个空文件。你可以试试用图形用户界面 GUI 能否这么快得达到效果！

我们用来创造这个奇迹的方法包含一个熟悉的命令 mkdir、一个奇异的 shell 花括号扩展以及一个新命令 touch。mkdir 命令结合-p 选项(-p 选项使 mkdir 命令按指定的路径创建父目录）的同时用花括号扩展，便完成了 100 个目录的创建。

touch 命令一般用于设定或是更新文件的修改时间。然而，当文件名参数是一个不存在的文件时，那么该命令就会创建一个空文件。

playground 文件夹里，总共创建了 100 个叫做 file-A 的文件。现在，我们可以查找它们。

```
me@linuxbox ~]$ find playground -type f -name 'file-A'
```

请注意，与 ls 命令不同，find 命令不会产生有排列顺序的结果，其输出顺序是由在存储设备中的布局决定的。下面的命令行验证了该文件夹确实有 100 个 file-A 文件。

```
[me@linuxbox ~]$ find playground -type f -name 'file-A' | wc -l
100
```

下面来看一个根据文件的修改时间查找文件的例子，这在创建备份文件以及按时间顺序排列文件时非常有用。首先需要创建一个用作比较修改时间的参照文件。

```
[me@linuxbox ~]$ touch playground/timestamp
```

该命令行创建了一个名为 timestamp 的空文件，并将当前时刻设为该文件的修改时间。我们可以使用另外一个便捷的命令 stat 来检验执行效果，stat 命令可以说是 ls 的增强版，该命令会将系统所掌握文件的所有信息及属性全部显示出来。

```
[me@linuxbox ~]$ stat playground/timestamp
  File: `playground/timestamp'
  Size: 0          Blocks: 0      IO Block: 4096 regular empty file
Device: 803h/2051d Inode: 14265061 Links: 1
Access: (0644/-rw-r--r--) Uid: ( 1001/ me)   Gid: ( 1001/ me)
Access: 2012-10-08 15:15:39.000000000 -0400
Modify: 2012-10-08 15:15:39.000000000 -0400
Change: 2012-10-08 15:15:39.000000000 -0400
```

当我们再一次对此文件执行 touch 命令并用 stat 命令检验时，会发现文件的时间得到了更新。

```
[me@linuxbox ~]$ touch playground/timestamp
[me@linuxbox ~]$ stat playground/timestamp
  File: `playground/timestamp'
  Size: 0          Blocks: 0      IO Block: 4096 regular empty file
Device: 803h/2051d Inode: 14265061 Links: 1
Access: (0644/-rw-r--r--) Uid: ( 1001/ me)   Gid: ( 1001/ me)
Access: 2012-10-08 15:23:33.000000000 -0400
Modify: 2012-10-08 15:23:33.000000000 -0400
Change: 2012-10-08 15:23:33.000000000 -0400
```

接下来，我们便可以用 find 命令更新 playground 文件夹里的一些文件。

```
[me@linuxbox ~]$ find playground -type f -name 'file-B' -exec touch '{}' ';'
```

该命令行更新了 playground 文件夹里面叫做 file-B 的所有文件。下面我们通过比较参照文件 timestamp 与其他文件的修改时间，使用 find 命令查找刚刚被更新的文件。

```
[me@linuxbox ~]$ find playground -type f -newer playground/timestamp
```

命令行的运行结果包含 100 个文件名为 file-B 的文件。由于我们是在对 timestamp 文件执行了 touch 命令之后，才对 playground 文件夹中对名为 file-B 的所有文件执行了 touch 操作，所以它们现在要比 timestamp 文件新，从而我们可选用-newer test 选项来查找。

最后，回顾之前讨论的查找不安全访问权限文件的例子，并将其用于 playground 目录。

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 \) -or \( -type d
-not -perm 0700 \)
```

该命令行列出了 playground 目录下的所有的 100 个子目录以及 2600 个文件（再加上 timestamp 文件和 playground 自身，总共 2702 个），之所以全部列出是因为 playground 里面没有一个文件满足“安全访问权限”的要求。运用前面所学 find 命令的 operator 选项和 action 选项的知识，我们可以在命令后面增加 action

参数选项来改变 playground 目录下文件或目录的访问权限。

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 -exec chmod 0600
'{}' ';' \) -or \( -type d -not -perm 0700 -exec chmod 0700 '{}' ';' \)
```

依据日常经验，大家可能会觉得用两条命令——分别针对目录和文件，比用这样一个长而复杂的命令容易得多。不过知道这一知识点总比不知道好，此处的重点是，要了解如何结合使用操作符选项与行为选项，来执行一些有用的任务。

17.2.4 option 选项

最后，我们谈一下 find 命令的 option 选项。option 选项用于控制 find 命令的搜索范围。在构成 find 命令的表达式时，它们可能包含在其他测试选项或行为选项之中。表 17-8 列出了最常用的 option 选项。

表 17-8 find 命令的 option 选项

选项	描述
-depth	引导 find 程序处理目录前先处理目录内文件。当指定-delete 操作时，该参数选项会自动调用
-maxdepth levels	当执行测试条件行为时，设置 find 程序陷入目录数的最大级别数
-mindepth levels	在应用测试条件和行为之前，设置 find 程序陷入目录数的最小级别数
-mount	引导 find 不去遍历挂载在其他文件系统上的目录
-noleaf	指导 find 程序不要基于“正在搜索类 UNIX 文件系统”的假设来优化它的搜索。当扫描 DOS/Windows 文件系统和 CD 时，会用到该选项

第18章 归档和备份

维护系统数据安全是计算机系统管理者的基本任务之一，及时创建系统文件的备份文件是维护系统数据安全的一种常用方法。即便对于非系统管理员，经常创建备份文件或是在设备之间、文件夹之间移动大文件集通常都是非常有益的。

本章会介绍一些用于管理文件集合的常用命令。

文件压缩程序：

- gzip：压缩和解压缩文件工具。
- bzip2：块排序文件压缩工具。

文件归档程序：

- tar：磁带归档工具。
- zip：打包和压缩文件。

文件同步程序：

- rsync：远程文件和目录的同步。

18.1 文件压缩

在计算领域的发展历史中，人们一直在努力实现以最小的可利用空间存储最多的数据，其中可利用空间包括内存、存储设备或者网络带宽。许多如今认为理所当然的数据服务，比如便携式音乐播放器、高清电视和宽带互联网等之所以能够存在，都应归功于有效的数据压缩技术。

数据压缩是一个移除数据冗余信息的过程。如下例，假设有一张 100 × 100 像素的全黑图像文件，就数据存储而言（假设每个像素占用 24 位，也就是 3 个字节），该图像也会占用 30,000（100 × 100 × 3 = 30,000）字节的存储量。

一幅只有一种颜色的图像包含了完全冗余的数据，我们要是聪明的话，编码该图像数据时可以直接简单地描述成有 30,000 个字节的黑色像素。因此，无需存储一个包含 30,000 字节的 0（图像文件里面，黑色通常用零表示）的数据块，而是将这些数据压缩成数字 30,000 和一个 0 来表示。这种数据压缩技术，称为游程编码（run-lengh encoding），它是最基本的一种压缩技术。现今的压缩技术则更先进、更复杂，但基本目标一直是消除冗余数据信息。

压缩算法（压缩采用的数学方法）一般分为两大类：无损压缩与有损压缩。无损压缩保留原文件中的所有数据，也就是说这种方式的压缩文件还原时，还原后的文件与原文件完全一致。而有损压缩，在压缩时为了实现更大程度的压缩而删除了某些数据信息，有损压缩文件还原时，与原文件并不是完全吻合，但是与原文件差别并不大。JPEG（图像压缩技术）和 MP3（音频压缩技术）技术是典型的有损压缩实例。下面的讨论中，仅仅涉及无损压缩，因为计算机上的大多数数据无法容忍任何数据损失。

18.1.1 gzip——文件压缩与解压缩

gzip 命令用于压缩一个或更多文件。执行命令后，原文件会被其压缩文件取代。与之相反，gunzip 命令则将压缩文件还原为原文件。示例如下。

```
[me@linuxbox ~]$ ls -l /etc > foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me    me    15738 2012-10-14 07:15 foo.txt
[me@linuxbox ~]$ gzip foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me    me    3230 2012-10-14 07:15 foo.txt.gz
[me@linuxbox ~]$ gunzip foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me    me    15738 2012-10-14 07:15 foo.txt
```

面本例中，我们首先创建了一个名为 foo.txt 的文本文件，其内容为某目录所含文件的列表清单，然后运行 gzip 命令。于是，压缩后的文件 foo.txt.gz 便取代了原文件。foo.*表达式的文件，我们可以看到原文件已被其压缩文件取代，并且压缩后的文件大小差不多才是原文件的 1/5。此外，我们还可以看出，压缩后的文件与原文件有着相同的权限和时间戳。

接着，我们运用 gunzip 命令进行解压缩，如此该压缩文件又被原始文件取代，而且权限和时间戳仍然保持一致。

gzip 有许多选项，表 18-1 列出了一些。

表 18-1 gzip 的选项

选项	功能描述
-c	将输出内容写到标准输出端口并且保持原有文件。也可以用--stdout 或是--to-stdout 替代
-d	解压缩。加上此选项，gzip 命令便类似于 gunzip。也可以用--decompress 或--uncompress 替代
-f	强制压缩，即便原文件的压缩版本已经存在了。也可以用--force 替代
-h	显示有用信息。也可以用--help 代替
-l	列出所有压缩文件的压缩统计。也可以用--list 代替
-r	如果该命令行的操作参数中有一个或是多个是目录，那么递归压缩包含在目录中的文件。也可以用--recursive 代替
-t	检验压缩文件的完整性。也可以用--test 代替
-v	在压缩时显示详细信息。也可以用--verbose 代替
-number	设定压缩级别。number 是 1（速度最快，压缩比最小）~9（速度最慢，压缩比最大）范围中的一个整数。当然 1~9 的数值亦可以分别描述为--fast 和--best。gzip 默认的压缩级别是 6

回顾前面的例子

```
[me@linuxbox ~]$ gzip foo.txt
[me@linuxbox ~]$ gzip -tv foo.txt.gz
foo.txt.gz:      OK
[me@linuxbox ~]$ gzip -d foo.txt.gz
```

此例中，首先，我们将压缩文件 foo.txt.gz 取代了原文件 foo.txt。接着，我们运用-t、-v 选项检查压缩文件的完整性。最后，解压缩该文件为原来的形式。

借助标准输入输出，gzip 有很多有趣的用法。

```
[me@linuxbox ~]$ ls -l /etc | gzip > foo.txt.gz
```

此命令创建了一个目录列表的压缩版本。

gunzip 命令用于解压 gzip 的压缩文件，并且默认解压缩后缀为“.gz”的文件，所以，我们没有必要明确指定，只要指定名与已存在的非压缩文件名不冲突就可以了。

```
[me@linuxbox ~]$ gunzip foo.txt
```

如果只是希望查看某个压缩文本文件的内容，可以直接输入下面的命令行：

```
[me@linuxbox ~]$ gunzip -c foo.txt | less
```

或者，利用 zcat 命令联合 gzip 一起，其效果等同于带有-c 选项的 gunzip。zcat 的功能与 cat 命令相同，只是它的操作对象是压缩文件。用 zcat 命令处理 gzip 压缩文件的示例如下。

```
[me@linuxbox ~]$ zcat foo.txt.gz | less
```

注意 同样也有 zless 命令，它与前面所讲的 less 的管道功能相同。

18.1.2 bzip2——牺牲速度以换取高质量的数据压缩

bzip2 程序由 Julian Seward 开发，与 gzip 命令功能相仿，但使用不同的压缩算法。该算法具有高质量的数据压缩能力，但却降低了压缩速度。多数情况下，其用法与 gzip 类似，只是用 bzip2 压缩后的文件以.bz2 为后缀。

```
[me@linuxbox ~]$ ls -l /etc > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-r--r-- 1 me    me    15738 2012-10-17 13:51 foo.txt
[me@linuxbox ~]$ bzip2 foo.txt
[me@linuxbox ~]$ ls -l foo.txt.bz2
-rw-r--r-- 1 me    me     2792 2012-10-17 13:51 foo.txt.bz2
[me@linuxbox ~]$ bunzip2 foo.txt.bz2
```

由此例可以看出，bzip2 用法与 gzip 类似，前面所讨论的 gzip 的所有选项（除了-r 选项），bzip2 都支持。然而，要注意的是，两者的压缩级别选项（-number）有些许不同。与此同时，解压缩 bzip2 压缩文件的专用工具是 bunzip2 和 bzcat 命令。

bzip2 还配有专门的 bzip2recover 命令，该命令用于恢复损坏的.bz2 文件。

不要强制压缩

有时，笔者看到有人试图压缩已经用有效压缩算法压缩过的文件，他们通常会这么做。

```
$ gzip picture.jpg
```

不可以这么做，这只是在浪费时间和空间而已！如果对一个已压缩文件再次进行压缩，实际上只会导致文件变大。因为所有的压缩技术会在压缩文件前首先增加一些开销，以描述本次压缩。如果试图压缩一个已无冗余信息的文件，那么此次压缩不会腾出任何空间来抵消增加的开销。

18.2 文件归档

归档是与压缩操作配合使用的一个常用文件管理任务。归档是一个聚集众多文件并将它们组合为一个大文件的过程，它通常作为系统备份的一部分，而且通常也用于将旧数据从某个系统移到某些长期存储设备的情况下。

18.2.1 tar——磁带归档工具

tar 命令是类 UNIX 系统中用于归档文件的经典工具。tar 是 tape archive 的缩写，由此可见，该命令最初始的作用就是磁带备份。虽然该命令仍可用于传统的磁带备份，但同样也可用于其他存储设备。大家肯定经常看到文件名以.tar 和.tgz 结尾的文件，它们分别是用普通的 tar 命令归档的文件和用 gzip 归档的文件。tar 归档文件可以由许多独立的文件、一个或多个目录层次或者两者的混合组合而成，其用法如下。

```
tar mode[options] pathname...
```

其中的 mode 是指表 18-2 中（此列表只列出了部分模式，想获得全部信息可以查看 tar 的 man 手册页）列出的操作模式的一种。

表 18-2 tar 命令的操作模式

模式	描述
c	创建文件和/或目录列表的归档文件
x	从归档文件中提取文件
t	在归档文件末尾追加指定路径名
r	列出归档文件的内容

tar 命令在选项的表达方式上有点奇怪，所以，我们需要举一些例子以说明其是如何工作的。首先，重新创建一个上一章中所建的 playground 文件夹。

```
[me@linuxbox ~]$ mkdir -p playground/dir-{00{1..9},0{10..99},100}
[me@linuxbox ~]$ touch playground/dir-{00{1..9},0{10..99},100}/file-{A..Z}
```

下面，用 tar 命令为整个 playground 文件夹创建一个归档文件。

```
[me@linuxbox ~]$ tar cf playground.tar playground
```

该命令行创建了一个叫做 playground.tar 的 tar 归档文件，该归档文件包含了 playground 文件夹的整个目录结构。从命令行中我们可以看到，tar 命令的操作模式 c 参数和用于指定归档文件名的 f 参数可以直接连着写在一起而中间不需要连字符隔开。然而，请注意，mode 参数必须在任何选项之前指定。

下面的命令行用于列出归档文件的内容，可以用于查看已经备份了哪些文件。

```
[me@linuxbox ~]$ tar tf playground.tar
```

如若想获取更详细的信息，可以增加-v（详细信息）选项。

```
[me@linuxbox ~]$ tar tvf playground.tar
```

现在，将 playground 文件夹中的内容提取到一个新的位置。首先，创建一个名为 foo 的新文件夹，然后切换工作目录，再提取该归档文件。

```
[me@linuxbox ~]$ mkdir foo
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground.tar
[me@linuxbox foo]$ ls
playground
```

查看~/foo/playground 目录下的内容，便会发现该归档文件已经成功提取，并且是原文件的精确复制。但是存在一个问题，除非是以超级用户的名义执行该命令，不然，从归档文件中提取出来的文件和目录的所有权属于执行归档操作的用户而不是文件的原始作者。

tar 命令处理档案文件路径名的方式也很有趣，其默认的路径名是相对路径而不是绝对路径，tar 命令创建归档文件时会简单地通过移除路径名前面的斜杠来实现相对路径。作为演示，下面会重新创建一个归档文件，此次明确指定一个绝对路径。

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ tar cf playground2.tar ~/playground
```

记住，当按下 Enter 键时，上面命令行中输入的目录~/playground 会扩展为/home/me/playground，也就是绝对路径。接下来，我们按照前面的步骤从归档文件中提取文件，注意观察所发生的变化。

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground2.tar
[me@linuxbox foo]$ ls
home   playground
[me@linuxbox foo]$ ls home
me
[me@linuxbox foo]$ ls home/me
playground
```

此刻，我们便会发现当解压第二个归档文件时，在当前工作目录~/foo 下重新创建了一个 home/me/playground 目录，而不是在认定的绝对路径根目录下创建的。这样的工作方式看起来很奇怪，但是却更有用，因为如此可以将归档文件解压缩到任何目录下而不用被迫解压到原目录下，使用-v 选项重复操作此实例可以详细地了解其运行情况。

下面让我们看一个假想的但很实用的 tar 命令应用实例。假设我们需要将一个系统上的主目录及其内容复制到另外一个系统上，并且具备用于实现这一转移过程的 USB 大硬盘。在现代 Linux 系统中，此硬盘会自动挂载到/media 目录下。再假设 USB 硬盘连接系统后，设备名为 BigDisk。接着用 tar 进行文件归档，示例如下。

```
[me@linuxbox ~]$ sudo tar cf /media/BigDisk/home.tar /home
```

tar 归档的文件写入硬盘后，我们将硬盘卸载，再将其与另外一台计算机连接。同样，此硬盘挂载在了/media/BigDisk 目录下。那么如何解压缩该归档文件，示例如下。

```
[me@linuxbox2 ~]$ cd /
[me@linuxbox2 /]$ sudo tar xf /media/BigDisk/home.tar
```

这里需要重点注意的是，我们首先要将工作目录改为根目录，以便文件解压缩在根目录下，因为归档文件中的所有文件采用的都是相对路径。

当从归档文件中提取文件时，可以限制只提取某些文件。例如，如果希望从归档文件中只提取单个文件，可以用如下命令行。

```
tar xf archive.tar  pathname
```

在命令后面添加要提取的文件的路径名，可以确保 tar 只恢复指定文件，而且可以指定多个路径名。注意，指定的路径名必须是存储在归档文件中的完整、

准确的相对路径。在指定路径名时，通常不支持通配符。但是，GNU 版本的 tar 命令（在 Linux 发行版本中该版本的 tar 最常见）通过使用--wildcards 选项而支持通配符。下面就是一个利用前面的 playground.tar 归档文件实践通配符的例子。

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground2.tar --wildcards 'home/me/playground/
dir-*/file-A'
```

此命令行只会提取那些路径名与通配符 dir-*匹配的文件。

tar 命令创建归档文件时通常辅助以 find 命令。首先使用 find 命令查找到需要被归档的文件，然后使用 tar 对这些文件进行归档，实例如下。

```
[me@linuxbox ~]$ find playground -name 'file-A' -exec tar rf playground.tar '{
}' '+'
```

上例，使用 find 命令匹配 playground 文件夹中叫做 file-A 的文件，然后借助-exec 操作选项，启动 tar 的附加模式 r 将匹配文件添加到归档文件 playground.tar 中。

tar 命令结合 find 命令很适合创建目录树以及整个系统的增量备份，使用 find 命令找到那些在时间戳文件之后创建的文件，便可以创建一份只包含上一次归档之后创建的文件的归档文件，当然假定该时间戳文件是在每一个归档文件创建之后就立刻更新。

tar 命令还可以利用标准输入输出。下面就是一个综合例子。

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ find playground -name 'file-A' | tar cf - --files-from=- | gzip
> playground.tgz
```

本例中，先用 find 命令搜索得到匹配文件列表，然后将匹配文件再送至 tar 命令处理。如果文件名前面明确指定有连字符“-”，那就意味着这是标准输入输出的文件（顺便讲一下，使用“-”代表标准“输入/输出”的惯例，其他许多程序也都采用）。--files-from 选项（也可以简写成-T）则指定了 tar 命令从文件中而不是从命令行中读取文件路径名列表。最后，tar 命令归档后的文件再送至 gzip 进行压缩，由此得到压缩归档文件 playground.tgz。后缀.tgz 已经惯例性成为经 gzip 压缩的 tar 归档文件名的后缀，当然，我们有时也用.tar.gz 作后缀。

虽然可以从外部使用 gzip 命令创建压缩归档文件，但现代 GNU 版本的 tar 命令则提供 gzip + z 选项和 bzip2+j 选项直接实现这一功能。以前面的例子为例，可以将命令行简化为以下命令行。

```
[me@linuxbox ~]$ find playground -name 'file-A' | tar czf playground.tgz -T -
```

当然，如果我们想要创建一个 bzip2 压缩的归档文件，可以这么做。

```
[me@linuxbox ~]$ find playground -name 'file-A' | tar cjf playground.tbz -T -
```

通过简单地将压缩选项从 z 变为 j（并将输出文件后缀改为.tbz 以显示是 bzip2 压缩的文件），即可实现 bzip2 式的压缩归档文件。

利用 tar 命令在系统之间传输网络文件，是 tar 另外一个利用标准输入输出的有趣用法。假设，有两台类 UNIX 系统的计算机正在运行，并且都安装了 tar 命令和 ssh 命令，于是，我们便可以将远程系统（本例中远程系统主机名叫做 remote-sys）中的某目录转移到本地系统。

```
[me@linuxbox ~]$ mkdir remote-stuff
[me@linuxbox ~]$ cd remote-stuff
[me@linuxbox remote-stuff]$ ssh remote-sys 'tar cf - Documents' | tar xf -
me@remote-sys's password:
[me@linuxbox remote-stuff]$ ls
Documents
```

上例中，名为 Documents 的目录从 remote-sys 的远程系统复制到本地系统上的 remote-stuff 的文件目录里。这是如何实现的呢？首先，用 ssh 程序在远程系统上启动 tar 命令，此时可能会联想到前面所讲的 ssh 具有在联网机器上运行远程程序并将结果显示在本地系统的能力，也就是远程系统的标准输出送至本地系统显示。于是，可以利用这一特性，我们可以将 tar 命令创建的归档文件（用 c 模式创建的）送至标准输出而不是直接输出文件（- f 选项），然后通过 ssh 建立的加密隧道将该归档文件送至本地系统。在本地系统上，我们再执行 tar 命令提取（x 模式）标准输入的归档文件（同样用 f 选项加上连字符作为参数）。

18.2.2 zip——打包压缩文件

zip 程序既是文件压缩工具也是文件归档工具。Windows 用户肯定很熟悉这种文件格式，因为其读写的是.zip 后缀的文件。然而，Linux 系统中，gzip 才是主要的压缩指令，而 bzip2 仅次之。Linux 用户主要使用 zip 程序与 Windows 系统交换文件，而不是将其用于压缩或是归档文件。

zip 最基本的调用方式如下。

```
zip options zipfile file...
```

例如，创建一个 playground 的 zip 归档文件，可以输入下面的命令行。

```
[me@linuxbox ~]$ zip -r playground.zip playground
```

此例中，如果不加-r 选项递归的话，只会保留 playground 这个目录而不包括目录中内容。虽然程序会自动默认添加后缀.zip，但为了以示清晰，最好还是在命令行中添加文件后缀。

zip 归档文件创建的过程中，zip 通常会显示如下的一系列信息。

```
adding: playground/dir-020/file-Z (stored 0%)
adding: playground/dir-020/file-Y (stored 0%)
adding: playground/dir-020/file-X (stored 0%)
adding: playground/dir-087/ (stored 0%)
adding: playground/dir-087/file-S (stored 0%)
```

这些信息显示的是每个新添归档文件的状态。zip 使用两种存储方式向归档文件中添加文件。第一，不对文件进行压缩直接存储，如本例；第二，缩小文件大小，即对文件进行压缩后存储。紧随存储方法之后显示的数值表示的是实现的压缩比。由于使用的 playground 文件夹是空文件夹，所以并没有对其内容进行压缩。

利用 unzip，我们可以直接提取 zip 文件中的内容。

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ unzip ../playground.zip
```

关于 zip，有一点需要注意（与 tar 命令相比），即如果指定的归档文件已经存在，那么 zip 仅仅只会更新而不会取而代之。这意味着原来存在的归档文件会保留下来，只是增加了一些新文件，原有匹配文件则被替换。

通过给 unzip 指定提取的文件名，我们可以选择性地从 zip 归档文件中提取文件。

```
[me@linuxbox ~]$ unzip -l playground.zip playground/dir-087/file-Z
Archive:  ./playground.zip
  Length     Date   Time    Name
 --------    ----   ----    ----
        0  10-05-12 09:25   playground/dir-087/file-Z
 --------                   -------
        0                   1 file
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ unzip ../playground.zip playground/dir-087/file-Z
Archive:  ../playground.zip
replace playground/dir-087/file-Z? [y]es, [n]o, [A]ll, [N]one, [r]ename: y
 extracting: playground/dir-087/file-Z
```

使用-l 选项，unzip 只会列出归档文件的内容而不会从中提取文件。如果没有指定任何文件，unzip 将会提取归档文件中的所有文件，我们可以增加-v 选项得到更详细的列表。注意当提取的文件与已存在文件冲突时，原文件被取代之

前会提示用户是否执行此替换操作。

与 tar 命令类似，zip 命令也可以利用标准输入输出，尽管此用法在某种程度上来说作用并不大。我们也可以用-@选项将多个文件送至 zip 进行压缩。

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ find playground -name "file-A" | zip -@ file-A.zip
```

本例中，我们利用 find 命令产生一个匹配-name 项的“file-A”文件列表，然后将结果直接作为 zip 命令的输入，从而得到了一个包含选定文件的归档文件 file-A.zip。

zip 同样可以将结果送至标准输出，但是由于只有极少的命令能够利用其输出结果，所以这种用法具有局限性。不幸的是，unzip 程序不支持标准输入，所以 zip 和 unzip 不能像 tar 命令一样一起用于处理网络文件。

然而，zip 命令支持标准输入，所以可以用于压缩其他程序的输出结果。

```
[me@linuxbox ~]$ ls -l /etc/ | zip ls-etc.zip -
 adding: - (deflated 80%)
```

本例中，我们将 ls 的输出结果列表直接送给 zip。与 tar 命令一样，zip 会默认末尾的连字符代表输入的文件是标准输入。

当指定-p 选项后，unzip 命令便将其输出结果以标准形式输出。

```
[me@linuxbox ~]$ unzip -p ls-etc.zip | less
```

到目前为止，本章只涉及了 zip 和 unzip 命令的一些基本用法，它们其实还有很多选项，尽管有些只适用于其他系统的特定平台，但是它们使用起来很灵活。zip 和 unzip 的 man 手册页都很全面，且包含了很多有用的例子。

18.3 同步文件和目录

将一个或多个目录与本地系统（通常是某种可移动存储设备）或是远程系统上其他的目录保持同步，是维护系统备份文件的常用方法。例如，本地系统上有一个正在开发的网站备份，用户通常会在远程 Web 服务器上进行“实时”备份以实现同步更新。

18.3.1 rsync——远程文件、目录的同步

针对类 UNIX 系统，完成这一同步任务最合适的工具当属 rsync。该命令通

过运用 rsync 远程更新协议，同步本地系统与远程系统上的目录，该协议允许 rsync 命令快速检测到本地和远程系统上两个目录之间的不同，从而以最少数量的复制动作以完成两个目录之间的同步。因此，rsync 命令与其他复制命令相比，显得既快又经济。

rsync 命令调用方式如下。

```
rsync options source destination
```

这里的 source 和 destination 是下列选项之一：

- 一个本地文件或目录；
- 一个远程文件或目录，形式为[user@]host:path；
- 一个远程 rsync 服务器，由 rsync://[user@]host[:port]/path 指定。

请注意，source 和 destination 中必须有一个本地文件，因为 rsync 不支持远程系统与远程系统之间的复制。

我们在本地文件上实践 rsync 命令，首先应清空 foo 目录。

```
[me@linuxbox ~]$ rm -rf foo/*
```

接着，同步 playground 目录和它在 foo 目录中的相应副本：

```
[me@linuxbox ~]$ rsync -av playground foo
```

此命令行中，我们运用了-a（用于归档——进行递归归档并保留文件属性）选项和-v（详细输出）选项在 foo 目录中生成了 playground 目录的镜像备份。此命令行运行过程中，我们会看到一系列文件和目录被复制。最后，显示如下的汇总信息，表示文件复制的总量。

```
sent 135759 bytes  received 57870 bytes  387258.00 bytes/sec
total size is 3230  speedup is 0.02
```

再次运行该命令行，会得到不同的结果。

```
[me@linuxbox ~]$ rsync -av playgound foo
building file list ... done

 sent 22635 bytes  received 20 bytes  45310.00 bytes/sec
total size is 3230  speedup is 0.14
```

请注意，此时并不会列出文件列表。因为 rsync 目录检测出~/playground 和~/foo/playground 两个文件夹之间并没有区别，因此不需要进行任何复制操作。如果 playground 目录中的某个文件被修改了，那么 rsync 会检测到该变化并且只

复制这个刚刚更新的文件。

```
[me@linuxbox ~]$ touch playground/dir-099/file-Z
[me@linuxbox ~]$ rsync -av playground foo
building file list ... done
playground/dir-099/file-Z
sent 22685 bytes  received 42 bytes  45454.00 bytes/sec
total size is 3230  speedup is 0.14
```

下面举一个实际的例子，回想前面讲解 tar 命令时所举的虚拟外部硬盘的例子。当该设备与系统连接时，再次挂载在/media/BigDisk 目录下，就可以进行系统备份了。首先，我们在外部硬盘上创建一个/backup 的目录，然后使用 rsync 命令将系统中最重要的内容复制到该外部设备。

```
[me@linuxbox ~]$ mkdir /media/BigDisk/backup
[me@linuxbox ~]$ sudo rsync -av --delete /etc /home /usr/local /media/BigDisk/
backup
```

在本例中，系统中的/etc、/home 和/usr/local 目录成功备份到了虚拟存储设备中，同时添加了-delete 选项以移除那些残留于备份设备中而源设备中已经不存在的文件（这一步骤在第一次备份时无关紧要，但在后续的复制操作中会起作用）。重复插入该外围设备并运行 rsync 命令，对于小型系统来说，这是一个持续备份的好（虽然不是完美的）方法。当然，如果此处使用别名会更方便。于是，我们可以定义一个别名，并将其添加到.bashrc 文件中，以提供该备份功能。

```
alias backup='sudo rsync -av --delete /etc /home /usr/local /media/BigDisk/backup'
```

现在所要做的就是插入外围设备，运行别名 backup 即可完成上述备份操作。

18.3.2 在网络上使用 rsync 命令

通过网络复制文件是 rsync 用法的另一个美妙之处。毕竟，rsync 命令名中的 r 其实指的是 remote（远程）。远程复制可以由以下两种方法中的任一种实现。

方法之一是针对于已安装了 rsync 命令以及诸如 ssh 等远程 shell 程序的系统。假定本地网络有另外一个具有足够可利用硬盘空间的系统，同时希望利用远程系统而非外部设备进行备份操作。假使远程系统已经有一个用于存放备份文件的/backup 目录，那么我们便可以直接运行下面的命令。

```
[me@linuxbox ~]$ sudo rsync -av --delete --rsh=ssh /etc /home /usr/local remote-
sys:/backup
```

此处命令行改动了两个地方。第一，增加了--rsh=ssh 选项，该选项告诉 rsync

使用 ssh 命令作为其远程 shell 命令。只有这样，我们才可以通过 SSH 的加密隧道安全地从本地系统向远程主机传输数据。第二，在 destination 路径名前指定了远程主机名（本例中的远程主机名是 remote-sys）。

方法之二，使用 rsync 服务器同步网络文件，通过配置 rsync 运行一个守护进程监听进来的同步请求。这种方法通常用于远程系统的镜像备份。例如，Red Hat 软件为发行其 Fedora 系统，需要维持一个正在开发的大的软件包库。对于软件测试员来说，在发行版的测试阶段创建这个大集合的备份是很重要的，因为库中的文件会频繁变动（每天会有多次改动），所以通过周期性的同步来维持本地文件镜像要比批量复制软件库更可取。Georgia Tech 就维护了其中一个库，可以使用本地复制工具 rsync 以及 Georgia Tech 的 rsync 服务器来创建该库的镜像备份。

```
[me@linuxbox ~]$ mkdir fedora-devel
[me@linuxbox ~]$ rsync -av -delete rsync://rsync.gtlib.gatech.edu/fedora-
linux-core/development/i386/os fedora-devel
```

本例中，使用 rsync 远程服务的 URI 是由协议（rsync://）、远程主机名（rsync.gtlib.gatech.edu）和库的路径名组成。

第19章 正则表达式

在下面几章，我们会讨论一些用于文本操作的工具。我们已经知道，文本数据在类UNIX系统中（比如Linux）扮演着非常重要的角色。但是，在领略这些工具强大的功能前，我们还是先看一下经常与这些工具的复杂用法相关联的技术——正则表达式。

前面我们已经接触过命令行提供的许多特性和工具，并且也遇到过一些相当神秘的shell特性及命令，比如shell扩展和引用、键盘快捷键和命令历史记录等，更不用提vi编辑器了。正则表达式也延续了这种传统，而且可以说是众多特性中最神秘的一个（该说法应该会持有争议）。当然，并不是说这些特性不值得大家花时间去学习。恰恰相反，熟练掌握这些用法会给人意想不到的效果，尽管它们的全部价值可能不会立即体现出来。

19.1 什么是正则表达式

简单地说，正则表达式是一种符号表示法，用于识别文本模式。在某种程

度上，它们类似于匹配文件和路径名时使用的 shell 通配符，但其用途更广泛。许多命令行工具和大多数编程语言都支持正则表达式，以此来解决文本操作方面的问题。然而，在不同的工具，以及不同的编程语言之间，正则表达式都会略有不同，这让事情进一步麻烦起来。方便起见，我们将正则表达式的讨论限定在 POSIX 标准中（它涵盖了大多数命令行工具），与许多编程语言（最著名的 Perl）不同，这些编程语言使用的符号集要更多一些。

19.2 grep——文本搜索

我们用来处理正则表达式的主要程序是 grep。grep 名字源于 “global regular expression print”，由此也可以看到，grep 与正则表达式有关。实际上，grep 搜索文本文件中与指定正则表达式匹配的行，并将结果送至标准输出。

目前为止，我们我们已经利用 grep 搜索了固定的字符串，如下所示：

```
[me@linuxbox ~]$ ls /usr/bin | grep zip
```

该命令行的作用是列出*/usr/bin* 目录下文件名包含 zip 字符串的所有文件。

grep 程序按照如下方式接受选项和参数。

```
grep [options] regex [file...]
```

其中字符串 *regex* 代表的是某个正则表达式。

表 19-1 列出了 grep 常用的选项。

表 19-1　grep 选项

选项	功能描述
-i	忽略大小写。不区分大写和小写字符，也可以用--ignore-case 指定
-v	不匹配。正常情况下，grep 会输出匹配行，而该选项可使 grep 输出不包含匹配项的所有行。也可以用--invert-match 指定
-c	输出匹配项数目（如果有-v 选项，那就输出不匹配项的数目）而不是直接输出匹配行自身。也可以用--count 指定
-l	输出匹配项文件名而不是直接输出匹配行自身。也可以用--files-with-matches 指定
-L	与-l 选项类似，但输出的是不包含匹配项的文件名。也可以用--files-without-match 指定
-n	在每个匹配行前面加上该行在文件内的行号。也可以用--line-number 指定
-h	进行多文件搜索时，抑制文件名输出。也可以用--no-filename 指定

为了更为全面地了解 grep，我们创建几个文本文件来进行搜索。

```
[me@linuxbox ~]$ ls /bin > dirlist-bin.txt
[me@linuxbox ~]$ ls /usr/bin > dirlist-usr-bin.txt
[me@linuxbox ~]$ ls /sbin > dirlist-sbin.txt
[me@linuxbox ~]$ ls /usr/sbin > dirlist-usr-sbin.txt
[me@linuxbox ~]$ ls dirlist*.txt
dirlist-bin.txt   dirlist-sbin.txt    dirlist-usr-sbin.txt
dirlist-usr-bin.txt
```

我们可以对文件列表执行简单搜索，如下所示。

```
[me@linuxbox ~]$ grep bzip dirlist*.txt
dirlist-bin.txt:bzip2
dirlist-bin.txt:bzip2recover
```

本例中，grep 命令会搜索所所有的文件，以查找字符串 bzip，并找到了两个匹配项，而且这两个匹配项都在文件 *dirlist-bin.txt* 里。如果我们只对包含匹配项的文件感兴趣而不是对匹配项本身感兴趣，可以指定-l 选项。

```
[me@linuxbox ~]$ grep -l bzip dirlist*.txt
dirlist-bin.txt
```

相反，如果那只想查看那些不包含匹配项的文件，则可以用如下命令行。

```
[me@linuxbox ~]$ grep -L bzip dirlist*.txt
dirlist-sbin.txt
dirlist-usr-bin.txt
dirlist-usr-sbin.txt
```

19.3 元字符和文字

虽然看起来不是很明显，但 grep 搜索一直都在使用正则表达式，尽管那些例子都很简单。正则表达式 bzip 用于匹配文本中至少包含 4 个字符、存在连续的按 b、z、i、p 顺序组成的字符串的行。字符串 bzip 中的字符都是文字字符（literal character），即它们只能与自身进行匹配。除了文字字符，正则表达式还可以包含用于指定更为复杂的匹配的元字符。正则表达式的元字符包括以下字符。

```
^ $ . [ ] { } - ? * + ( ) | \
```

其他所有字符则被当作文字字符，但是在极少数的情况下，反斜杠字符用来创建元序列，以及用来对元字符进行转义，使其成为文字字符，而再被解释为元字符。

注意 可以看到，当 shell 在执行扩展时，许多正则表达式的元字符在 shell 中具有特殊的含义。所以，在命令行中输入包含元字符的正则表达式时，应把这些元字符用引号括起来以避免不必要的 shell 扩展。

19.4 任意字符

接下来讨论的第一个元字符是“点”字符或者句点字符，该字符用于匹配任意字符。如果将其加进某个正则表达式中，它将会在对应位置匹配任意字符。下面就是一个应用实例。

```
[me@linuxbox ~]$ grep -h '.zip' dirlist*.txt
bunzip2
bzip2
bzip2recover
gunzip
gzip
funzip
gpg-zip
preunzip
prezip
prezip-bin
unzip
unzipsfx
```

上述命令行，搜索到了所有匹配正则表达式.zip 的命令行。但其输出结果有一些有趣的地方，比如说输出中并没有包含 zip 程序，这是因为正则表达式中的“.”元字符将匹配长度增加到了 4 个字符。而“zip”只包含了 3 个字符，所以不匹配。同样，如果列表中某个文件包含了文件扩展名“.zip”，那么该文件也会被认为是匹配文件，因为文件扩展名中的“.”符号也被当作“任意字符”处理了。

19.5 锚

插入符（^）和美元符号（$）在正则表达式被当做锚，也就是说正则表达式只与行的开头（^）或是末尾（$）的内容进行匹配比较。

```
[me@linuxbox ~]$ grep -h '^zip' dirlist*.txt
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
[me@linuxbox ~]$ grep -h 'zip$' dirlist*.txt
gunzip
gzip
funzip
gpg-zip
preunzip
prezip
```

```
unzip
zip
[me@linuxbox ~]$ grep -h '^zip$' dirlist*.txt
zip
```

上例中搜索的是行开头、行末尾都有字符串“zip”（例如 zip 自成一行）的文件。请注意，正则表达式“^$”（行开头和行末尾之间没有字符）将会匹配空行。

纵横填字字谜助手

我的妻子很喜欢玩纵横填字字谜这个游戏，并且有时她会拿一个具体问题向我寻求帮助。诸如“有一个 5 个字母的单词，它的第三个字母是 j，最后一个字母是 r，请问这是什么单词？”等等，这类问题不得不让我思考。

你们知道 Linux 系统本身自带一个字典吗？查看*/usr/share/dict* 目录，就会发现一个或是多个这样的字典。字典中的文件包含的是一个常常的单词列表，每个单词占一行，以字母表的顺序排列。在我自己的系统上，单词文件中包含了超过 98,500 个单词。可以输入如下命令行，得到上述字谜问题的可能答案：

```
[me@linuxbox ~]$ grep -i '^..j.r$' /usr/share/dict/words
Major
major
```

利用这样的正则表达式，便可以找到字典中所有匹配的单词，即满足字符长度为 5、第三个字母是 *j* 以及末尾字母是 *r* 的所有单词。

19.6　中括号表达式和字符类

中括号除了可以用于匹配正则表达式中给定位置的任意字符外，还可以用于匹配指定字符集中的单个字符。借助于中括号，我们可以指定要匹配的字符集（也包括那些可能会被解释为元字符的字符）。如下命令行则利用了一个两个字母组成的字符集，用于匹配包含 bzip 或 gzip 字符串的文本行。

```
[me@linuxbox ~]$ grep -h '[bg]zip' dirlist*.txt
bzip2
bzip2recover
gzip
```

一个字符集可以包含任意数目的字符，并且当元字符放置到中括号中时，会失去它们的特殊含义。然而，在两种情况下，则会在中括号中使用元字符，并且会有不同的含义。第一个就是插入符（^），它在中括号内使用表示否定；

另外一个是连字符（-），它表示字符范围。

19.6.1 否定

如果中括号内的第一个字符是插入符（^），那么剩下的字符则被当作不应该在指定位置出现的字符集。作为演示，我们对前面的例子稍作修改。

```
[me@linuxbox ~]$ grep -h '[^bg]zip' dirlist*.txt
bunzip2
gunzip
funzip
gpg-zip
preunzip
prezip
prezip-bin
unzip
unzipsfx
```

通过使用否定操作，我们可以得到那些包含 zip 字符串但 zip 前面既不是 *b* 也不是 *g* 的所有程序。请注意，此时 *zip* 命令仍然没有出现在结果列表中，由此可见否定，字符集仍然需要在指定位置有对应字符，只不过这个字符不是否定字符集中的成员而已。

插入符号“^”只有是中括号表达式中的第一个字符时才会被当作否定符，如果不是第一个，“^”将会丧失其特殊含义而成为普通字符。

19.6.2 传统字符范围

如果我们希望建立一个正则表达式，用于查找文件名以大写字母开头的文件，可以用下面的命令行。

```
[me@linuxbox ~]$ grep -h '^[ABCDEFGHIJKLMNOPQRSTUVWXZY]' dirlist*.txt
```

这仅仅是将 26 个大写字母写入中括号的小事，但是要输入 26 个字母实在有点麻烦，我们可以用下面的简单方法完成。

```
[me@linuxbox ~]$ grep -h '^[A-Z]' dirlist*.txt
MAKEDEV
ControlPanel
GET
HEAD
POST
X
X11
Xorg
MAKEFLOPPIES
NetworkManager
NetworkManagerDispatcher
```

通过使用三个字符表示的字符范围，我们可以缩写这 26 个字母。能够按照这种方式表达的任何字符范围可以包含多个范围，比如下面这个表达式可以匹配以字母和数字开头的所有文件名。

```
[me@linuxbox ~]$ grep -h '^[A-Za-z0-9]' dirlist*.txt
```

在字符范围中，可以看到连字符有了特殊的用法，那么如何在中括号中真正包括一个连字符字符？可将连字符作为中括号内的第一个字符，示例如下。

```
[me@linuxbox ~]$ grep -h '[A-Z]' dirlist*.txt
```

此命令行匹配的是文件名中包含一个大写字母的文件。而下面的命令行

```
[me@linuxbox ~]$ grep -h '[-AZ]' dirlist*.txt
```

匹配的则是文件名包含连字符、大写字母 *A* 或大写字母 *Z* 的文件。

19.6.3 POSIX 字符类

传统的字符范围表示方法很容易理解，而且能够有效、快速地指定字符集。但不足之处在于，它并不是所有情况都适用。虽然到目前为止，在使用 grep 命令时还没有遇到过任何问题，但是在其他程序中则可能会遇到问题。

在第 4 章，我们讨论了如何使用通配符来执行路径名扩展。在该讨论中我们提到，字符范围的用法几乎与其在正则表达式中一致，现在问题出现了。

```
[me@linuxbox ~]$ ls /usr/sbin/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]*
/usr/sbin/MAKEFLOPPIES
/usr/sbin/NetworkManagerDispatcher
/usr/sbin/NetworkManager
```

Linux 发行版本不同，上述命令行得到的结果可能会不同，甚至有可能是空列表。本例中的列表来自于 Ubuntu 系统。该命令行得到了预期效果——只有以大写字母开头的文件列表。但是，如果我们使用下面的命令行，便会得到完全不同的结果（只显示了输出结果的一部分）。

```
[me@linuxbox ~]$ ls /usr/sbin/[A-Z]*
/usr/sbin/biosdecode
/usr/sbin/chat
/usr/sbin/chgpasswd
/usr/sbin/chpasswd
/usr/sbin/chroot
/usr/sbin/cleanup-info
/usr/sbin/complain
/usr/sbin/console-kit-daemon
```

为什么会出现这样的差异？说来话长，简单解释如下。

在 UNIX 开发初期，它只识别 ASCII 字符，而正是这一特性导致了上面的差异。在 ASCII 码中，前 32 个字符（第 0~31 字符）都是控制字符（像 Tab 键、空格键以及 Enter 键等），后 32 个字符（第 32~63）包含可打印字符，包括大多数的标点符号以及数字 0~9，接下来的 32 个（第 64~95）包含大写字母和一些标点符号，最后的 31 个（第 96~127）则包含小写字母以及更多的标点符号。基于这样的安排，使用 ASCII 的系统使用了下面这种排序：

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

这与通常的字典顺序不一样，字典中字母的顺序表通常如下。

```
aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ
```

随着 UNIX 在美国以外国家的普及，人们越来越希望计算机能支持美式英语中找不到的字符。于是，ASCII 字符表也得以扩展，开始使用 8 位二进制来表示，这也就增加了第 188～255 的字符，兼容了更多的语言。为了支持这种功能，POSIX 标准引入了域（locale）的概念，它通过不停调整以选择特定的位置所需要的字符集。我们可以使用下面的命令行查看系统的语言设置。

```
[me@linuxbox ~]$ echo $LANG
en_US.UTF-8
```

有了这个设置，POSIX 兼容的应用程序使用的便是字典中的字母排列顺序，而不是用 ASCII 码中的字符排列顺序。这样，便解释了上面命令行的诡异行为。A~Z 的字符范围，用字典中的顺序诠释时，包括了字母表中除了小写字母 a 的所有字母，因此使用命令行 ls /usr/sbin/[A-Z]*才会出现全然不同的结果。

为了解决这一问题，POSIX 标准包含了许多标准字符类，这些字符类提供了一些有用的字符范围，见表 19-2。

表 19-2　POSIX 字符类

字符类	描述
[:alnum:]	字母字符和数字字符；在 ASCII 码中，与[A-Za-z0-9]等效
[:word:]	基本与[:alnum:]一样，只是多了一个下划线字符(_)
[:alpha:]	字母字符；在 ASCII 中，等效于[A-Za-z]
[:blank:]	包括空格和制表符
[:cntrl:]	ASCII 控制码；包括 ASCII 字符 0~31 以及 127
[:digit:]	数字 0~9
[:graph:]	可见字符；在 ASCII 中，包括字符 33~126

续表

字符类	描述
[:lower:]	小写字母
[:punct:]	标点符号字符；在 ASCII 中，与[-!"#$%&'()*+,./:;<=>?@[\\\]_`{|}~]等效
[:print:]	可打印字符；包括[:graph:]中的所有字符再加上空格字符
[:space:]	空白字符如空格符、制表符、回车符、换行符、垂直制表符以及换页符。在 ASCII 中，等效为[\t\r\n\v\f]
[:upper:]	大写字母
[:xdigit:]	用于表示十六进制的字符；在 ASCII 中，与[0-9A-Fa-f]等效

当然，即便是有了这么多字符类，仍然没有比较方便的方法表示部分范围，如[A-M]。

使用字符类，我们可以重复上述大写字母的例子，并得到改善的输出结果。

```
[me@linuxbox ~]$ ls /usr/sbin/[[:upper:]]*
/usr/sbin/MAKEFLOPPIES
/usr/sbin/NetworkManagerDispatcher
/usr/sbin/NetworkManager
```

然而，请记住，上述并不是一个正则表达式的示例，它其实是 shell 路径名扩展的一个例子。在此处提及，主要是因为这两种用法都支持 POSIX 字符类。

恢复为传统的排列顺序

你可以设置自己的系统采用传统的（ASCII）字符顺序，方法就是改变 LANG 环境变量的值。LANG 变量包含语言的名称以及该语言环境中使用的字符集，该参数值在安装 Linux 系统选择安装语言时就已经设定。

要查看环境设置，使用下面的 locale 命令。

```
[me@linuxbox ~]$ locale
LANG=en_US.UTF-8
LC_CTYPE="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_PAPER="en_US.UTF-8"
LC_NAME="en_US.UTF-8"
LC_ADDRESS="en_US.UTF-8"
LC_TELEPHONE="en_US.UTF-8"
LC_MEASUREMENT="en_US.UTF-8"
LC_IDENTIFICATION="en_US.UTF-8"
LC_ALL=
```

将环境设置为使用传统的 UNIX 行为，可将 LANG 变量值设为 POSIX:

```
[me@linuxbox ~]$ export LANG=POSIX
```

请注意，这样的改变将会导致系统使用美式英语（更精确地说，是 ASCII 码格式）的字符集，所以在进行此改变之前要三思。

如果想永久性维持该变化，可以在系统.bashrc 文件中添加下面的命令行。

```
export LANG=POSIX
```

19.7 POSIX 基本正则表达式和扩展正则表达式的比较

在读者正觉得正则表达式已经复杂得不能再复杂时，又会发现 POSIX 规范将正则表达式的实现方法分为了两种：基本正则表达式（BRE）和扩展正则表达式（ERE）。到目前为止，我们所讨论的正则表达式的所有特性，都得到了兼容 POSIX 的应用程序的支持，并且都是以 BRE 的方式实现。grep 命令就是这样的一个例子。

BRE 和 ERE 到底有什么区别？其实仅仅是元字符的不同！在 BRE 方式中，只承认^ 、$、 . 、[、] 、*这些是元字符，所有其他的字符都被识别为文字字符。而 ERE 中，则添加了(、) 、{ 、} 、?、 +|、等元字符（及其相关功能）。

然而（也是有趣的部分），只有在用反斜杠进行转义的情况下，字符(、)、{、}才会在 BRE 被当作元字符处理，而 ERE 中，任何元符号前面加上反斜杠反而会使其被当作文字字符来处理。

由于下面要讨论的特性是 ERE 的一部分，所以需要使用不一样的 grep。传统上，这是由 egrep 程序来执行的，但是 GNU 版本的 grep 可以运用-E 选项以支持 ERE 方式。

POSIX

在 20 世纪 80 年代，UNIX 成为一款非常受欢迎的商业操作系统，但是直到 1988 年，UNIX 世界仍然一片混乱。许多电脑制造商从 UNIX 的创造者 AT&T 获得了 UNIX 源代码授权，并且都发行了不同版本的操作系统产品。然而，制造商在努力追求产品差异化的同时，每个制造商都增加了自己专用的变化以及扩展，这就渐渐限制了软件的兼容性。由于一直由专有厂商销售，所以每一个厂商都想尽办法锁定它们的客户群。UNIX 史上的这段黑暗时期被大家称之为“割据时代”。

接着，我们进入了 IEEE（Institute of Electrical and Electronics Engineers）时代。在 20 世纪 80 年代中期，IEEE 开始开发一套规范 UNIX 和类 UNIX 系统工作方式的标准。这些标准，官方名称是 IEEE 1003，定义了应用程序接口

（API）、shell 以及一些实用程序，它们可以在标准类 UNIX 系统中找到。该标准由 Richard Stallman 提议命名为 POSIX，它是 Portable Operating System Interface（末尾增加 *X* 只是为了更流畅）的缩写，后来被 IEEE 采纳。

19.8 或选项

我们将要讨论的第一个扩展正则表达式的特性是或选项（alternation），它是用于匹配表达式集的工具。括号表达式可以从指定字符集中匹配单一字符，而或选项则用于从字符串集或正则表达式集中寻找匹配项。

以下便利用 grep 结合 echo 作为演示实例。首先，我们进行一个简单的字符串匹配。

```
[me@linuxbox ~]$ echo "AAA" | grep AAA
AAA
[me@linuxbox ~]$ echo "BBB" | grep AAA
[me@linuxbox ~]$
```

这是一个非常直白的例子，将 echo 的输出结果送至 grep 进行匹配搜索。如果匹配成功，结果便输出打印出来；如无匹配项，则无结果输出。

现在添加或选项，它用元字符“|”表示。

```
[me@linuxbox ~]$ echo "AAA" | grep -E 'AAA|BBB'
AAA
[me@linuxbox ~]$ echo "BBB" | grep -E 'AAA|BBB'
BBB
[me@linuxbox ~]$ echo "CCC" | grep -E 'AAA|BBB'
[me@linuxbox ~]$
```

这里出现了'AAA|BBB'正则表达式，此表达式的含义是“匹配字符串 AAA 或者匹配字符串 BBB”。请注意，由于此处使用的是扩展特性，所以 grep 增加了-E 选项（虽然可以使用 egrep 命令来代替），并且将正则表达式用引号引起来以防止 shell 将元字符“|”当作管道操作符来处理。另外，或选项并不局限于两种选择，还可以有更多的选择项。

```
[me@linuxbox ~]$ echo "AAA" | grep -E 'AAA|BBB|CCC'
AAA
```

为了将或选项可与其他正则表达式符号结合使用，我们可以用“()”将或选项的所有元素与其他符号隔开。

```
[me@linuxbox ~]$ grep -Eh '^(bz|gz|zip)' dirlist*.txt
```

以上表达式的含义是匹配文件名以 bz、gz 或是 zip 开头的文件。如果不使用括号“()”，该正则表达式的含义就完全不同，其匹配的便是文件名以 bz 开头或者是包含 gz 和 zip 的文件。

```
[me@linuxbox ~]$ grep -Eh '^bz|gz|zip' dirlist*.txt
```

19.9 限定符

扩展正则表达式（ERE）提供多种方法指定某元素匹配的次数。

19.9.1 ?——匹配某元素 0 次或 1 次

该限定符实际上意味着“前面的元素可选”。比如，如果我们想检查某电话号码的有效性。所谓电话号码有效，指的是电话号码必须是下面两种形式 *(nnn)nnn-nnnn* 和 *nnn nnn-nnnn* 中的一种，其中 n 是数值。于是，我们可以构造如下所示的正则表达式。

```
^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
```

此表达式中，括号字符的后面增加了“？”符号以表示括号字符只能匹配一次或零次。同样，由于括号字符在 ERE 中通常是元字符，所以其前面加上了反斜杠告诉 shell 此括号为文字字符。

示例如下。

```
[me@linuxbox ~]$ echo "(555) 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)? [0-9]
[0-9][0-9]$'
(555) 123-4567
[me@linuxbox ~]$ echo "555 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)? [0-9]
[0-9][0-9]-[0-9][0-9][0-9][0-9]$'
555 123-4567
[me@linuxbox ~]$ echo "AAA 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)? [0-9]
[0-9][0-9]-[0-9][0-9][0-9][0-9]$'
[me@linuxbox ~]$
```

由此可以看出，该表达式匹配了上述两种形式的电话号码，但不匹配那些包含非数字字符的号码。

19.9.2 *——匹配某元素多次或零次

与“？”元字符类似，“*”用于表示一个可选择的条目。然而，与“?”不同，该条目可以多次出现，而不仅仅是一次。例如，如果我们想知道一串字符是否是一句话，也就是说，这串字符是否以大写字母开头而以句号结束，并且中间

内容是任意数目的大小写字母和空格，那么要匹配这种非常粗糙的句子定义，可以用如下正则表达式。

```
[[:upper:]][[:upper:][:lower:] ]*\.
```

该表达式包含了三个条目：包含[:upper:]字符类的中括号表达式、包含[:upper:]和[:lower:]两个字符类以及一个空格的中括号表达式、用反斜杠转义过的圆点符号。第二个条目后面紧跟着“*”元字符，所以只要句子的第一个字母是大写字母，后面不管会出现多少数目的大小写字母都无关紧要。

```
[me@linuxbox ~]$ echo "This works." | grep -E '[[:upper:]][[:upper:][:lower:]
 ]*\.'
This works.
[me@linuxbox ~]$ echo "This Works." | grep -E '[[:upper:]][[:upper:][:lower:]
 ]*\.'
This Works.
[me@linuxbox ~]$ echo "this does not" | grep -E '[[:upper:]][[:upper:][:lower:
] ]*\.'
[me@linuxbox ~]$
```

该表达式匹配前两个测试语句，但是不匹配第三个，原因是它的首字母不是大写并且末尾没有句号。

19.9.3 +——匹配某元素一次或多次

“+”元字符与“*”非常类似，只是“+”要求置于其前面的元素至少出现一次。示例如下，该正则表达式用于匹配由单个空格分隔的一个或者多个字母字符组成的行。

```
^([[:alpha:]]+ ?)+$
```

示例如下。

```
[me@linuxbox ~]$ echo "This that" | grep -E '^([[:alpha:]]+ ?)+$'
This that
[me@linuxbox ~]$ echo "a b c" | grep -E '^([[:alpha:]]+ ?)+$'
a b c
[me@linuxbox ~]$ echo "a b 9" | grep -E '^([[:alpha:]]+ ?)+$'
[me@linuxbox ~]$ echo "abc  d" | grep -E '^([[:alpha:]]+ ?)+$'
[me@linuxbox ~]$
```

我们可以看到此表达式并不匹配“a b 9”这一行，因为该行包含了非字母字符“9”，同样也不匹配“abc d”这一行，因为 c 和 d 之间被多个空格符分开了。

19.9.4 {}——以指定次数匹配某元素

“{”和“}”元字符用于描述最小和最大次数的需求匹配。可以通过 4 种方

法来指定，见表 19-3。

表 19-3 指定匹配次数

指定项	含义
{*n*}	前面的元素恰好出现 *n* 次则匹配
{*n*,*m*}	前面的元素出现的次数在 *n*~*m* 次之间时则匹配
{*n*,}	前面的元素出现次数超过 *n* 次则匹配
{,*m*}	前面的元素出现次数不超过 *m* 次则匹配

回到前面电话号码的例子，我们可以运用此处讲到的指定重复次数的办法将原来的正则表达式

```
“^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$”
```

简化为

```
“^\(?[0-9]{3}\)? [0-9]{3}-[0-9]{4}$”
```

示例如下：

```
[me@linuxbox ~]$ echo "(555) 123-4567" | grep -E '^\(?[0-9]{3}\)? [0-9]{3}-[0-
9]{4}$'
(555) 123-4567
[me@linuxbox ~]$ echo "555 123-4567" | grep -E '^\(?[0-9]{3}\)? [0-9]{3}-[0-9]
{4}$'
555 123-4567
[me@linuxbox ~]$ echo "5555 123-4567" | grep -E '^\(?[0-9]{3}\)? [0-9]{3}-[0-9
]{4}$'
[me@linuxbox ~]$
```

结果表明，修改后的表达式不管有无括号都可验证数字的有效性，并剔除那些格式不正确的数字。

19.10 正则表达式的应用

让我们回顾一些前面已经用过的命令，观察它们如何使用正则表达式。

19.10.1 用 grep 命令验证号码簿

前面的例子中，我们只验证了一个电话号码的有效性，而检验一个号码列表往往才是实际需求，所以我们先创建一个号码列表。于是，我们在命令行中输入一些“神奇的咒语”以创建号码列表，之所以称其为“神奇”是因为里面多数命令到目前为止本书还未涉及到。不过不用担心，我们将在下面的章节中

会详细讲述。下面的代码便是神奇咒语的内容。

```
[me@linuxbox ~]$ for i in {1..10}; do echo "(${RANDOM:0:3}) ${RANDOM:0:3}-$
{RANDOM:0:4}" >> phonelist.txt; done
```

该命令行会产生一个包含 10 个电话号码的名为 *phonelist.txt* 的文件。每次重复该命令行，该列表就会添加 10 个号码。我们也可以通过更改命令行前端的数字 10 来指定创建更多或更少的电话号码。然而，如果检查文件内容，我们会发现如下问题。

```
[me@linuxbox ~]$ cat phonelist.txt
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
```

其中有部分数字是畸形的，而这正好满足我们的练习要求，因为接下来就是要用 grep 判断它们的有效性。

进行有效验证，可对文件内容进行扫描以搜索无效数字，并将结果显示出来。

```
[me@linuxbox ~]$ grep -Ev '^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$' phonelist.txt
(292) 108-518
(129) 44-1379
[me@linuxbox ~]$
```

此处我们使用了-v 选项输出相反结果，也就是输出列表中不符合指定表达式的那些行。此表达式本身在每行末尾使用了锚元字符，从而确保每一个数字末尾没有多余的字符。另外，本表达式要求有效的号码中必须要有括号，与前面所举的号码例子有稍许不同。

19.10.2 用 find 查找奇怪文件名的文件

find 命令的 test 选项可以用正则表达式表示。运用正则表达式时，find 和 grep 有一点不同，grep 命令是搜索那些只包含与指定表达式匹配的字符串的行，而 find 则要求文件名与指定表达式完全一致。下面的例子中，我们将利用 find 命令结合正则表达式来查找文件名中不包含如下所示集合中的任一字符的文件。

```
[-_./0-9a-zA-Z]
```

用此表达式进行搜索，将会输出文件名中包含内嵌空格以及其他潜在不规范字符的文件。

```
[me@linuxbox ~]$ find . -regex '.*[^-_./0-9a-zA-Z].*'
```

由于要求整个路径名与正则表达式的描述完全一致，所以表达式的两端增加了“.*”以匹配 0 个或多个字符。在表达式的中间部分，我们使用了一个否定的中括号表达式，其中包含了可接受的路径名字符集。

19.10.3 用 locate 查找文件

locate 命令既支持基本正则表达式（--regexp 选项），也支持扩展正则表达式（--regex 选项）。利用 locate，可以完成之前对 dirlist 文件所做的许多操作。

```
[me@linuxbox ~]$ locate --regex 'bin/(bz|gz|zip)'
/bin/bzcat
/bin/bzcmp
/bin/bzdiff
/bin/bzegrep
/bin/bzexe
/bin/bzfgrep
/bin/bzgrep
/bin/bzip2
/bin/bzip2recover
/bin/bzless
/bin/bzmore
/bin/gzexe
/bin/gzip
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

利用或选项，我们搜索到了那些路径名中包含 bin/bz、bin/gz 或/bin/zip 等字符串的文件。

19.10.4 利用 less 和 vim 命令搜索文本

less 和 vim 采用同样的方法进行文本搜索。按下“/”键后输入正则表达式，即开始进行搜索。我们可以利用 less 查看 *phonelist.txt* 文件的内容：

```
[me@linuxbox ~]$ less phonelist.txt
```

然后查找有效的表达式。

```
(232) 298-2265
(624) 381-1078
```

```
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
~
~
~
/^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$
```

less 会以高亮的方式显示匹配字符传，这样就很容易找到那些无效的号码。

```
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
~
~
~
(END)
```

另一方面，vim 也支持基本正则表达式，所以，我们可以用下面的搜索表达式。

```
/([0-9]\{3\}) [0-9]\{3\}-[0-9]\{4\}
```

可以看到，表达式基本一样。然而，在扩展表达式中被当作元字符的许多字符在，在基本表达中则被看作文字字符，只有使用反斜杠转义后才会被当作元字符。匹配项是否以高亮形式显示，则取决于自己系统上 vim 的设置。如果没有，请尝试在命令模式中输入 hlsearch 以激活高亮搜索。

注意　由于 Linux 发行版本的不同，vim 可能不支持文本高亮搜索。尤其是 Ubuntu，其默认提供了一个 vim 的精简版本。在这样的系统上，可能会需要用软件包管理器来安装一个较完整的 vim 版本。

19.11　本章结尾语

本章介绍了正则表达式的很多用法。当然，如果用它们进行一些额外的应用搜索，你会发现正则表达式有更多的用途。我们可以通过查看 man 手册页获

取更详细的内容。

```
[me@linuxbox ~]$ cd /usr/share/man/man1
[me@linuxbox man1]$ zgrep -El 'regex|regular expression' *.gz
```

zgrep 程序比 grep 在前端多了一个字母 z，由此便可以对压缩文件进行搜索。本例中，我们在 man 文件通常存放的目录下对压缩的第一部分 man 手册页进行了搜索。该命令行的输出结果是一列包含字符串 regex 或 regular expression 的文件。可以看到，正则表达式可用于大量应用程序中。

本章未谈及基本正则表达式的另外一个特性——后向引用（*back reference*），该特性将在下一章中进行详解。

第20章 文本处理

由于所有类 UNIX 操作系统都严重依赖于文本文件来进行某些数据类型的存储，所以需要有很多可以进行文本操作的工具。本章主要介绍一些与“切割”文本有关的命令，第 21 章会进一步探讨文本处理工具，并重点讲解那些用于格式化文本输出以及其他满足人类需求的程序。

本章首先会回顾之前讲过的一些命令，然后讲解一些新的命令。

- cat：连接文件并打印到标准输出。
- sort：对文本行排序。
- uniq：报告并省略重复行。
- cut：从每一行中移除文本区域。
- paste：合并文件文本行。
- join：基于某个共享字段来联合两个文件的文本行。

- comm：逐行比较两个已经排好序的文件。
- diff：逐行比较文件。
- patch：对原文件打补丁。
- tr：转换或删除字符。
- sed：用于过滤和转换文本的流编辑器。
- aspel：交互式拼写检查器。

20.1 文本应用程序

到目前为止，我们总共介绍了两种文本编辑器（nano 和 vim），看过一堆配置文件，并且目睹了许多命令的输出都是文本格式。那么，除了这些，文本操作还有哪些用途？事实证明，它还有很大用途。

20.1.1 文件

许多人都采用纯文本格式编辑文件。虽然大家都知道用一些较小的文本文件进行一些简单的笔记很方便、很实用，但同样，我们也可以用文本格式编辑较大的文档。有一种常用的方法，即首先在文本编辑器中编辑大型文档的内容，然后使用标记语言描述文件格式。许多科学论文就是用这种方式撰写的，因为基于 UNIX 的文本处理系统是最早一批支持先进排版布局的，而这些先进的排版技术又正是技术领域的专家们所需要的。

20.1.2 网页

网页可以说是世界上最常见的电子文档。网页属于文本文档，一般使用 HTML（Hypertext Markup Language）或者 XML（eXtensible Markup Language）等标记语言描述内容的可视化形式。

20.1.3 电子邮件

电子邮件本质上是一种基于文本的媒介，即便是非文本附件，在传输的时候也会被转成文本格式。读者可以亲自验证这一点，首先下载一个电子邮件信息并用 less 命令查看其内容，就会发现其内容包含一个 header 开头，其描述的是此封邮件的来源及其在传输过程中所经历的处理，然后才是邮件信

息的正文。

20.1.4　打印机输出

在类 UNIX 系统中，准备向打印机传送的信息是以纯文本格式传送的。如果该页包含图像，则将其转换成 PostScript 文本格式页面描述语言后再送至指定程序以打印图像像素。

20.1.5　程序源代码

类 UNIX 系统中的许多命令行程序都是为了支持系统管理和软件开发而编写的，文本处理程序也不例外。它们中多数是为解决软件开发问题而设计的。文本处理对软件开发者如此重要，是因为所有的软件都是从文本开始，程序员实际所编写的源代码，也总是以文本的形式编辑。

20.2 温故以求新

在第 6 章，我们学习了一些既支持命令行参数输入也支持标准输入的命令。不过当时只是泛泛而谈，现在我们将详细讨论这些命令如何用于文本处理。

20.2.1　cat——进行文件之间的拼接并且输出到标准输出

cat 命令有许多有趣的参数选项，而其中多数则是用于提高文本内容的可视化效果。-A 选项就是一个例子，它用于显示文本中的非打印字符。例如，用户有时会想知道可见文本中是否嵌入了控制字符，其中最为常见的就是制表符（而不是空格）以及回车符，在 MS-DOS 风格的文本文件中，回车符经常作为结束符出现。另一种常见情况是文件中包含末尾带有空格的文本行。

我们创建一个测试文件，用 cat 程序作为一个简单的文字处理器。为此，只需要输入 cat 命令（随后指定了用于重定向输出的文件）再输入文本内容，按 Enter 键结束行输入，最后按下 Ctrl-D 告诉 cat 到达文件末尾。下例中，我们输入了一个以 Tab 制表符开头、空格符结尾的文本行。

```
[me@linuxbox ~]$ cat > foo.txt
The quick brown fox jumped over the lazy dog.
[me@linuxbox ~]$
```

下面，我们利用带有-A 选项的 cat 命令显示文本内容：

```
[me@linuxbox ~]$ cat -A foo.txt
^IThe quick brown fox jumped over the lazy dog. $
[me@linuxbox ~]$
```

输出结果表明，文本中的 Tab 制表符由符号“^I”表示。这是一种常见的表示方法，意思是“Ctrl-I”，结果证明，它等同于 Tab 制表符。同时，在文件末尾出现的“$”符说明行末尾存在空格。

MS-DOS 文本与 UNIX 文本的比较

我们利用 cat 查找文本中的非打印字符，原因之一是 cat 可以发现隐藏的回车符。而这些隐藏的回车符来自哪里呢？当然是 DOS 和 Windows！UNIX 和 DOS 在文本文件中定义每行结束的方式并不相同，UNIX 以换行符（ASCII 码 10）作为行末尾，而 MS-DOS 系统及其衍生系统则使用回车符（ASCII 码 13）和换行符共同作为行末尾。

有几种方法可以将文件从 DOS 格式转化为 UNIX 格式。许多 Linux 系统，都自带 dos2UNIX 和 UNIX2dos 程序，它们用于 DOS 和 UNIX 格式之间的相互转换。然而，即便系统中没有 dos2UNIX 程序也没关系，DOS 格式转换为 UNIX 格式的过程非常简单，只要将多余的回车符删除就可以，此过程可以用本章后续所讲的一些程序很简单地实现。

cat 也有很多用于修改文本的参数选项。最著名的两个选项：-n，对行编号；-s，禁止输出多个空白行。示例如下。

```
[me@linuxbox ~]$ cat > foo.txt
The quick brown fox

jumped over the lazy dog.
[me@linuxbox ~]$ cat -ns foo.txt
    1 The quick brown fox
    2
    3 jumped over the lazy dog.
[me@linuxbox ~]$
```

本例中，我们创建了一个 foo.txt 测试文件的新版本，该文件内容为两个文本行，并以空白行隔开。用 cat 加-ns 选项对其操作后，多余的空白行便被移除，并对剩余的行进行了编号。然而这并不是多个进程在操作这个文本，只有一个进程。

20.2.2 sort——对文本行进行排序

sort 是一个排序程序，它的操作对象为标准输入或是命令行中指定的一个或

多个文件后将结果送至标准输出。与 cat 用法类似，如下所示，我们将直接使用键盘演示标准输入内容的处理过程。

```
[me@linuxbox ~]$ sort > foo.txt
c
b
a
[me@linuxbox ~]$ cat foo.txt
a
b
c
```

输入 sort 命令后，输入字母 c、b、a，最后按下 Ctrl-D 结束输入。然后查看处理结果，会发现这些行都以排好的顺序出现。

由于 sort 命令允许多个文件作为其输入参数，所以可以将多个文件融合为一个已排序的整体文件。例如，我们有三个文本文件，并期望将它们拼接为一个已排序的整体文件。我们可以用下面的命令行去执行。

```
sort file1.txt file2.txt file3.txt > final_sorted_list.txt
```

sort 有一些有趣的选项。表 20-1 列出了部分。

表 20-1 常见的 sort 选项

选项	全局选项表示	描述
-b	--ignore-leading-blanks	默认情况下，整个行都会进行排序操作，也就是从行的第一个字符开始。添加该选项后，sort 会忽略行开头的空格，并且从第一个非空白字符开始排序
-f	--ignore-case	排序时不区分字符大小写
-n	--numeric-sort	基于字符串的长度进行排序。该选项使得文件按数值顺序而不是按字母表顺序进行排序
-r	--reverse	逆序排序。输出结果按照降序排列而不是升序
-k	--key=*field1*[,*field2*]	对 *field1* 与 *field2* 之间的字符排序，而不是整个文本行
-m	--merge	将每个输入参数当作已排好序的文件名。将多个文件合并为一个排好序的文件，而不执行额外的排序操作
-o	--output=*file*	将排序结果输出到文件而不是标准输出
-t	--field-separator=*char*	定义字段分隔符。默认情况下，字段是由空格或制表符分开的

尽管以上多数选项的作用都易从其字面意义中看出，但也有一些例外，用于数值排序的-n 选项就是一个例子。该参数选项可使 sort 根据数值进行排序。作为演示，我们可将 du 命令的输出结果进行排序，以确定最大的硬盘空间用户。正常情况下，du 命令会列出一个以路径名顺序排列的列表。

```
[me@linuxbox ~]$ du -s /usr/share/* | head
252             /usr/share/aclocal
96              /usr/share/acpi-support
8               /usr/share/adduser
196             /usr/share/alacarte
344             /usr/share/alsa
8               /usr/share/alsa-base
12488           /usr/share/anthy
8               /usr/share/apmd
21440           /usr/share/app-install
48              /usr/share/application-registry
```

本例中，我们把结果管道到 head 命令，把输出结果限制为只显示前 10 行。我们能够产生一个按数值排序的列表，来显示 10 个最大的空间消费者。

```
[me@linuxbox ~]$ du -s /usr/share/* | sort -nr | head
509940          /usr/share/locale-langpack
242660          /usr/share/doc
197560          /usr/share/fonts
179144          /usr/share/gnome
146764          /usr/share/myspell
144304          /usr/share/gimp
135880          /usr/share/dict
76508           /usr/share/icons
68072           /usr/share/apps
62844           /usr/share/foomatic
```

通过使用-nr 参数选项，我们便可以产生一个逆向的数值排序，它使得最大数值排列在第一位。这种排序起作用是因为数值出现在每一行的开头。但是如果我们想要基于文本行中的某个数值排序，又会怎样呢？例如，命令 ls-l 的输出结果：

```
[me@linuxbox ~]$ ls -l /usr/bin | head
total 152948
-rwxr-xr-x 1 root   root     34824 2012-04-04 02:42 [
-rwxr-xr-x 1 root   root    101556 2011-11-27 06:08 a2p
-rwxr-xr-x 1 root   root     13036 2012-02-27 08:22 aconnect
-rwxr-xr-x 1 root   root     10552 2011-08-15 10:34 acpi
-rwxr-xr-x 1 root   root      3800 2012-04-14 03:51 acpi_fakekey
-rwxr-xr-x 1 root   root      7536 2012-04-19 00:19 acpi_listen
-rwxr-xr-x 1 root   root      3576 2012-04-29 07:57 addpart
-rwxr-xr-x 1 root   root     20808 2012-01-03 18:02 addr2line
-rwxr-xr-x 1 root   root    489704 2012-10-09 17:02 adept_batch
```

此刻，忽略 ls 命令自有的根据文件大小排序的功能，而用 sort 程序依据文件大小进行排序。

```
[me@linuxbox ~]$ ls -l /usr/bin | sort -nr -k 5 | head
-rwxr-xr-x 1 root   root   8234216 2012-04-07 17:42 inkscape
-rwxr-xr-x 1 root   root   8222692 2012-04-07 17:42 inkview
-rwxr-xr-x 1 root   root   3746508 2012-03-07 23:45 gimp-2.4
-rwxr-xr-x 1 root   root   3654020 2012-08-26 16:16 quanta
-rwxr-xr-x 1 root   root   2928760 2012-09-10 14:31 gdbtui
```

```
-rwxr-xr-x 1 root   root   2928756 2012-09-10 14:31 gdb
-rwxr-xr-x 1 root   root   2602236 2012-10-10 12:56 net
-rwxr-xr-x 1 root   root   2304684 2012-10-10 12:56 rpcclient
-rwxr-xr-x 1 root   root   2241832 2012-04-04 05:56 aptitude
-rwxr-xr-x 1 root   root   2202476 2012-10-10 12:56 smbcacls
```

sort 的许多用法都与表格数据处理有关，比如上面 ls 命令的输出结果。如果我们把数据库这个术语应用到上面的表格中，我们会说每一行就是一项记录，而每一个记录又包含多个字段，诸如文件属性、链接数、文件名、文件大小等。sort 能够处理独立的字段，在数据库术语中，我们可以指定一个或多个关键字段作为排序的关键值。在上面的例子中，指定了 n 和 r 选项进行数值的逆序排序，并指定-k 5 让 sort 程序使用第 5 个字段作为排序的关键值。

k 这个参数选项非常有趣，并且有很多特性，但是首先我们需要了解 sort 是如何定义字段的。让我们考虑一个非常简单的文本文件，它只有一行，并且该行只包含了该作者的名字。

```
William   Shotts
```

默认情况下，sort 程序会把该行看作有两个字段。第一个字段包含“William”字符串，第二个字段则是“Shotts”。这意味着空白字符（空格和制表符）用作字段之间的界定符，并且在排序时，这些界定符是包括在字段中的。

重新回到前面的 ls 例子，我们可以看到 ls 的输出行包含 8 个字段，并且第 5 个字段指的是文件大小。

```
-rwxr-xr-x 1 root   root   8234216 2012-04-07 17:42 inkscape
```

让我们考虑用下面的文件。该文件包含从 2006 年～2008 年三款流行的 Linux 发行版的发行历史。文件每行都有 3 个字段：发行版本名、版本号和 MM/DD/YYYY 格式的发行日期。

```
SUSE        10.2      12/07/2006
Fedora      10        11/25/2008
SUSE        11.0      06/19/2008
Ubuntu      8.04      04/24/2008
Fedora      8         11/08/2007
SUSE        10.3      10/04/2007
Ubuntu      6.10      10/26/2006
Fedora      7         05/31/2007
Ubuntu      7.10      10/18/2007
Ubuntu      7.04      04/19/2007
SUSE        10.1      05/11/2006
Fedora      6         10/24/2006
Fedora      9         05/13/2008
Ubuntu      6.06      06/01/2006
Ubuntu      8.10      10/30/2008
Fedora      5         03/20/2006
```

使用文本编辑器（可能是 vim），输入此数据并将其保存为 *distros.txt*。

接下来，我们试着对该文件进行排序并观察其输出结果。

```
[me@linuxbox ~]$ sort distros.txt
Fedora          10        11/25/2008
Fedora          5         03/20/2006
Fedora          6         10/24/2006
Fedora          7         05/31/2007
Fedora          8         11/08/2007
Fedora          9         05/13/2008
SUSE            10.1      05/11/2006
SUSE            10.2      12/07/2006
SUSE            10.3      10/04/2007
SUSE            11.0      06/19/2008
Ubuntu          6.06      06/01/2006
Ubuntu          6.10      10/26/2006
Ubuntu          7.04      04/19/2007
Ubuntu          7.10      10/18/2007
Ubuntu          8.04      04/24/2008
Ubuntu          8.10      10/30/2008
```

嗯，大部分正确。Fedora 的版本号排序时却出现了问题。因为字符集中，字符 1 是在字符 5 前面的，所以导致版本 10 位于第一行而版本 9 却在最后。

为了解决这个问题，我们必须依据多个键值进行排序。首先我们对第一个字段进行字母排序，然后再对第三字段进行数值排序。sort 支持-k 选项的多个实例，所以可以指定多个排序键值。事实上，一个键值可能是一个字段范围，如果没有指定任何范围（如前面所举的例子），sort 会使用一个键值，该键值始于指定的字段，一直扩展到行尾。

如下便是采用多键值进行排序的语法。

```
[me@linuxbox ~]$ sort --key=1,1 --key=2n distros.txt
Fedora          5        03/20/2006
Fedora          6        10/24/2006
Fedora          7        05/31/2007
Fedora          8        11/08/2007
Fedora          9        05/13/2008
Fedora          10       11/25/2008
SUSE            10.1     05/11/2006
SUSE            10.2     12/07/2006
SUSE            10.3     10/04/2007
SUSE            11.0     06/19/2008
Ubuntu          6.06     06/01/2006
Ubuntu          6.10     10/26/2006
Ubuntu          7.04     04/19/2007
Ubuntu          7.10     10/18/2007
Ubuntu          8.04     04/24/2008
Ubuntu          8.10     10/30/2008
```

虽然为了清晰，我们使用了选项的长格式，但是-k1、-k 2n 格式是等价的。

在第一个 key 选项的实例中，指定了一个字段范围。因为我们只想对第一个字段排序，所以指定了“1，1”，它意味着“始于并且结束于第一个字段”。在第二个实例中，我们指定了 2n，表示“第二个字段是排序的键值，并且按照数值排序”。一个选项字母可能包含在一个键值说明符的末尾，用来指定排序的种类。这些选项字母与 sort 命令的全局选项一样：b（忽略开头空白字符）、n（数值排序）、r（逆序排序）等。

以上列表的第三个字段包含的日期形式并不利于排序。计算机中，日期通常以 YYYY-MM-DD 的形式存储，以方便按时间顺序排序，但该文本中的时间则是以美国形式 MM/DD/YYYY 存储。那么，该如何对其进行时间排序呢？

幸好 sort 提供了一种解决方法。sort 的 key 选项允许在字段中指定偏移，所以我们可以在字段内定义键值。

```
[me@linuxbox ~]$ sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt
Fedora          10      11/25/2008
Ubuntu          8.10    10/30/2008
SUSE            11.0    06/19/2008
Fedora          9       05/13/2008
Ubuntu          8.04    04/24/2008
Fedora          8       11/08/2007
Ubuntu          7.10    10/18/2007
SUSE            10.3    10/04/2007
Fedora          7       05/31/2007
Ubuntu          7.04    04/19/2007
SUSE            10.2    12/07/2006
Ubuntu          6.10    10/26/2006
Fedora          6       10/24/2006
Ubuntu          6.06    06/01/2006
SUSE            10.1    05/11/2006
Fedora          5       03/20/2006
```

通过指定-k 3.7，我们告诉 sort 从第三个字段的第 7 个字符开始排序，也就是从年份开始排序。同样，指定-k 3.1 和-k 3.4 选项以区分日期中的月和日，另外我们利用了 n、r 选项进行逆序数值排序。同时添加的 b 选项用来删除日期字段中开头的空格（行与行之间的空格字符数量不同，因此会影响排序结果）。

有些文件并不是使用制表符或空格符作为字段定界符，例如这个/etc/passwd 文件。

```
[me@linuxbox ~]$ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
```

```
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
```

该文件的字段之间以冒号（:）作为分界符，那么该如何利用关键字段对此文件进行排序呢？sort 提供了-t 选项定义字段分隔符，根据 passwd 文件的第 7 个字段内容进行排序（用户默认的 shell 环境），如下面的命令行。

```
[me@linuxbox ~]$ sort -t ':' -k 7 /etc/passwd | head
me:x:1001:1001:Myself,,,:/home/me:/bin/bash
root:x:0:0:root:/root:/bin/bash
dhcp:x:101:102::/nonexistent:/bin/false
gdm:x:106:114:Gnome Display Manager:/var/lib/gdm:/bin/false
hplip:x:104:7:HPLIP system user,,,:/var/run/hplip:/bin/false
klog:x:103:104::/home/klog:/bin/false
messagebus:x:108:119::/var/run/dbus:/bin/false
polkituser:x:110:122:PolicyKit,,,:/var/run/PolicyKit:/bin/false
pulse:x:107:116:PulseAudio daemon,,,:/var/run/pulse:/bin/false
```

指定冒号字符作为字段分隔符，便实现了依据第 7 个字段进行排序的目的。

20.2.3 uniq——通知或省略重复的行

与 sort 相比，uniq 算是一个轻量级命令。uniq 执行的是一个看似简单的任务，给定一个已排好序的文件（包括标准输入）后，uniq 会删除任何重复的行并将结果输出到标准输出中。它通常与 sort 结合使用以删除 sort 输出内容中重复的行。

注意 虽然 uniq 是一个与 sort 一起使用的传统的 UNIX 工具，但是 GNU 版本的 sort 同样支持-u 选项，以用于移除 sort 输出内容中的重复行。

创建一个文本文件以验证此特性。

```
[me@linuxbox ~]$ cat > foo.txt
a
b
c
a
b
c
```

不要忘了按下 Ctrl-D 以结束标准输入。此刻，如果运行 uniq，文件内容并没有很大改动，重复的行也并没有被删除。

```
[me@linuxbox ~]$ uniq foo.txt
a
b
c
a
```

```
b
c
```

uniq 只有对已经排好序的文本才有作用。

```
[me@linuxbox ~]$ sort foo.txt | uniq
a
b
c
```

这是因为 uniq 只能移除相邻的重复行。

当然，uniq 有许多选项，表 20-2 列出了常用的一些。

表 20-2　常见的 uniq 选项

选项	功能描述
-c	输出重复行列表，并且重复行前面加上其出现的次数
-d	只输出重复行，而不包括单独行
-f n	忽略每行前 n 个字段。字段之间以空格分开，这与 sort 类似，但与 sort 不同的是，uniq 没有提供参数设置可选择的字段分隔符
-i	行与行之间比较时忽略大小写
-s n	跳过（忽略）每行的前 n 个字符
-u	仅输出不重复的行。该选项是默认的

使用 uniq 的-c 选项，可输出文本中重复行的数量，示例如下。

```
[me@linuxbox ~]$ sort foo.txt | uniq -c
      2 a
      2 b
      2 c
```

20.3　切片和切块

下面讨论的 3 个命令，它们的作用是剥离文本文件的列，并将它们以期望的方式重组。

20.3.1　cut——删除文本行中的部分内容

cut 命令用于从文本行中提取一段文字并将其输出至标准输出。它可以接受多个文件和标准输入作为输入参数

指定所要提取的内容，实现起来确实有点别扭，表 20-3 列出的是指定时要用到的选项。

表 20-3 cut 选择选项

选项	功能描述
-c char_list	从文本行中提取 char_list 定义的部分内容。此列表可能会包含一个或更多冒号分开的数值范围
-f field_list	从文本行中提取 field_list 定义的一个或多个字段。该列表可能会包含由冒号分隔的一个、多个字段或字段范围
-d delim_char	指定-f 选项后，使用 delim_char 作为字段分界符。默认时，字段必须以单个 Tab 制表符隔开
--complement	从文本中提取整行，除了那些由-c 和/或-f 指定的部分

正如大家所看到的，cut 提取文本的方式非常不灵活。cut 适合从其他命令的输出结果中提取文本内容，而不是直接从输入文本中提取。我们可以判断下面的 distros.txt 文件是否达到了 cut 的提取要求，利用 cat 的-A 选项，可以检查该文件是否用 Tab 作为字段分隔符的。

```
[me@linuxbox ~]$ cat -A distros.txt
SUSE^I10.2^I12/07/2006$
Fedora^I10^I11/25/2008$
SUSE^I11.0^I06/19/2008$
Ubuntu^I8.04^I04/24/2008$
Fedora^I8^I11/08/2007$
SUSE^I10.3^I10/04/2007$
Ubuntu^I6.10^I10/26/2006$
Fedora^I7^I05/31/2007$
Ubuntu^I7.10^I10/18/2007$
Ubuntu^I7.04^I04/19/2007$
SUSE^I10.1^I05/11/2006$
Fedora^I6^I10/24/2006$
Fedora^I9^I05/13/2008$
Ubuntu^I6.06^I06/01/2006$
Ubuntu^I8.10^I10/30/2008$
Fedora^I5^I03/20/2006$
```

看起来不错——没有内嵌空格，字段之间仅仅只有单个的制表符。鉴于文件使用的是制表符而不是空格，所以可以使用-f 选项提取字段内容

```
[me@linuxbox ~]$ cut -f 3 distros.txt
12/07/2006
11/25/2008
06/19/2008
04/24/2008
11/08/2007
10/04/2007
10/26/2006
05/31/2007
10/18/2007
04/19/2007
05/11/2006
10/24/2006
05/13/2008
```

```
06/01/2006
10/30/2008
03/20/2006
```

由于 *distros* 文件是以制表符作为分界符的，所以用 cut 提取字段而不是字符再合适不过。这是因为用 Tab 作为分界符的文件，一般每行不会包含相同的字符数，所以计算字符在行内的位置很困难或是根本不可能。然而在上例中，我们已经提取好了包含相同长度数据的字段，从而可以拿此字段作为字符提取实例。

```
[me@linuxbox ~]$ cut -f 3 distros.txt | cut -c 7-10
2006
2008
2008
2008
2007
2007
2006
2007
2007
2007
2006
2006
2008
2006
2008
2006
```

对输出结果再进行一次 cut 操作，便可以将字段中与年份相对应的第 7 至第 10 个字符提取出来，命令行中的符号“7-10”指范围。cut 的 man 手册页包含了关于范围指定的完整描述。

当处理字段时，我们可以指定非 Tab 字符作为分界符。如下所示的例子演示的是从*/etc/passwd* 文件中提取了每行的第一个字段。

```
[me@linuxbox ~]$ cut -d ':' -f 1 /etc/passwd | head
root
daemon
bin
sys
sync
games
man
lp
mail
news
```

此处我们还使用了-d 选项指定冒号作为字段的分界符。

扩展制表符 Tab

上例中的 *distros.txt* 文件已经被格式化为 cut 提取最理想的形式。但是如

果想要一个可以完全由 cut 借助字符，而非字段执行提取操作的文件，该怎么办呢？这就需要我们将文件中的 Tab 符号置换成等长的空格符号。幸运的是，GNU 的 coreutils 软件包提供了 expand 这样一个工具，该命令的输入参数既可以是标准输入也可以是一至多个文件，改动后的文本会以标准形式输出。

如果事先用 expand 处理 distros.txt 文件，便可以直接用 cut -c 从文件中提取任意范围的字符。示例如下，通过 expand 扩展该文件后，使用 cut 提取文本行中从第 23 个到末尾的字符，这样，我们便单独提取出了发行年份这一列表栏。

```
[me@linuxbox ~]$ expand distros.txt | cut -c 23-
```

另外，coreutils 软件包同样也提供了 unexpand 命令，从而用空格取代了制表符。

20.3.2 paste——合并文本行

paste 命令是 cut 的逆操作，它不是从文本文件中提取列信息，而是向文件中增加一个或是更多的文本列。该命令读取多个文件并将每个文件中提取出的字段结合为一个整体的标准输出流。与 cut 类似，paste 也可以接受多个文件输入参数和标准输入。至于 paste 是如何运行的，我们可以通过下面的例子进行了解。如下所示，我们描述了一个使用 paste 对 *distros.txt* 文件按照发行版本的时间顺序而排序的例子。

首先，我们用前面所学的 sort 命令，得到一个依据日期进行排序的 distros 列表，并将输出结果储存于文件 *distros-by-date.txt* 中。

```
[me@linuxbox ~]$ sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt > distros-by-
date.txt
```

接下来，我们使用 cut 提取文件中前两个字段（distros 的名字和发行版本），并将结果存于文件 distro-versions.txt 中。

```
[me@linuxbox ~]$ cut -f 1,2 distros-by-date.txt > distros-versions.txt
[me@linuxbox ~]$ head distros-versions.txt
Fedora          10
Ubuntu          8.10
SUSE            11.0
Fedora          9
Ubuntu          8.04
Fedora          8
Ubuntu          7.10
SUSE            10.3
Fedora          7
Ubuntu          7.04
```

最后一步准备工作就是提取发行版本日期，并将结果存于 *distro-dates.txt* 文

件中。

```
[me@linuxbox ~]$ cut -f 3 distros-by-date.txt > distros-dates.txt
[me@linuxbox ~]$ head distros-dates.txt
11/25/2008
10/30/2008
06/19/2008
05/13/2008
04/24/2008
11/08/2007
10/18/2007
10/04/2007
05/31/2007
04/19/2007
```

此刻，万事俱备，只欠东风。作为整个例子的最后一步，我们使用 paste 将提取的日期这一列内容置于 distro 中操作系统名称和发行版本号这两列之前，于是便生成了一个按时间顺序排列的发行版本列表。这一过程只是简单地使用 paste 命令来将各参数按照指定顺序进行排列。

```
[me@linuxbox ~]$ paste distros-dates.txt distros-versions.txt
11/25/2008	Fedora	10
10/30/2008	Ubuntu	8.10
06/19/2008	SUSE	11.0
05/13/2008	Fedora	9
04/24/2008	Ubuntu	8.04
11/08/2007	Fedora	8
10/18/2007	Ubuntu	7.10
10/04/2007	SUSE	10.3
05/31/2007	Fedora	7
04/19/2007	Ubuntu	7.04
12/07/2006	SUSE	10.2
10/26/2006	Ubuntu	6.10
10/24/2006	Fedora	6
06/01/2006	Ubuntu	6.06
05/11/2006	SUSE	10.1
03/20/2006	Fedora	5
```

20.3.3 join——连接两文件中具有相同字段的行

从某种程度上来说，join 与 paste 类似，因为它也是向文件增加列信息，只是实现方式有些许不同。“join”操作通常与“关联数据库”联系在一起，它在关联数据库中把共享关键字段多个表格的数据组合成一个期望结果。“join”是一个基于共享关键字段将多个文件的数据拼接在一起的操作。

至于“join”命令在关联数据库中是如何操作的，我想大家一定想知道。示例如下，假定有一个比较小的包含两个表格的数据库，并且每个表格都只包含一项纪录。第一个表格，叫做 CUSTOMERS，它有三个字段，分别是顾客号码（CUSTNUM），顾客的姓（FNAME）以及顾客的名（LNAME）。

```
CUSTNUM         FNAME       LNAME
=========       ======      ======
4681934         John        Smith
```

第二个表格叫做 ORDERS，包含 4 个字段，它们是订单号（ORDERNUM）、顾客号码（CUSTNUM）、数量（QUAN）以及订购内容（ITEM）。

```
ORDERNUM        CUSTNUM     QUAN    ITEM
==========      =========   =====   ====
3014953305      4681934     1       Blue Widget
```

请注意，两个表格拥有公共字段 CUSTNUM。这很重要，因为这就是两个表格之间的联系。

join 操作完成了两个表格中的字段结合，从而得到了有用的输出结果，诸如用于准备一张发票。利用两个表格中 CUSTNUM 字段的共有匹配值，进行 join 操作后便会输出以下结果。

```
FNAME           LNAME       QUAN    ITEM
======          ======      =====   ====
John            Smith       1       Blue Widget
```

做为演示，我们需要创建两个具有共有字段的文件，于是便可以使用 *distros-by-date.txt* 文件。利用该文件，可以生成两个附属文件，其中一个文件包含的内容是发行时间（作为本例中的共有字段）和发行版本名。

```
[me@linuxbox ~]$ cut -f 1,1 distros-by-date.txt > distros-names.txt
[me@linuxbox ~]$ paste distros-dates.txt distros-names.txt > distros-key-names
.txt
[me@linuxbox ~]$ head distros-key-names.txt
11/25/2008      Fedora
10/30/2008      Ubuntu
06/19/2008      SUSE
05/13/2008      Fedora
04/24/2008      Ubuntu
11/08/2007      Fedora
10/18/2007      Ubuntu
10/04/2007      SUSE
05/31/2007      Fedora
04/19/2007      Ubuntu
```

第二个文件的内容则包含发行时间和发行版本号。

```
[me@linuxbox ~]$ cut -f 2,2 distros-by-date.txt > distros-vernums.txt
[me@linuxbox ~]$ paste distros-dates.txt distros-vernums.txt > distros-key-
vernums.txt
[me@linuxbox ~]$ head distros-key-vernums.txt
11/25/2008      10
10/30/2008      8.10
06/19/2008      11.0
05/13/2008      9
```

```
04/24/2008        8.04
11/08/2007        8
10/18/2007        7.10
10/04/2007        10.3
05/31/2007        7
04/19/2007        7.04
```

此刻，两个具有公共字段（“发行时间”作为共有字段）的文件便准备妥当。此处需要重点强调的是，文件必须事先依据共有关键字段排好序，因为只有这样 join 才能正常工作。

```
[me@linuxbox ~]$ join distros-key-names.txt  distros-key-vernums.txt | head
11/25/2008 Fedora 10
10/30/2008 Ubuntu 8.10
06/19/2008 SUSE 11.0
05/13/2008 Fedora 9
04/24/2008 Ubuntu 8.04
11/08/2007 Fedora 8
10/18/2007 Ubuntu 7.10
10/04/2007 SUSE 10.3
05/31/2007 Fedora 7
04/19/2007 Ubuntu 7.04
```

同样请注意，默认情况下，join 会把空格当作输入字段的分界符，而以单个空格作为输出字段的分界符，当然我们也可以通过指定参数选项改变这一默认属性。我们可以查看 join 的 man 手册页获取更详细信息。

20.4　文本比较

比较文本文件的版本号通常很有用，尤其对系统管理者以及软件开发者而言。例如，一个系统的管理者可能需要将已存在的配置文件与原先的配置文件相比较，以检查出系统漏洞，与此类似，程序员也会经常需要查看程序代码所经历的变化。

20.4.1　comm——逐行比较两个已排序文件

comm 命令一般用于文本文件之间的比较，显示两文件中相异的行以及相同的行。作为演示，我们首先利用 cat 生成两个近乎一样的文本文件。

```
[me@linuxbox ~]$ cat > file1.txt
a
b
c
d
[me@linuxbox ~]$ cat > file2.txt
b
```

```
c
d
e
```

接下来，便利用 comm 比较两个文件的差异。

```
[me@linuxbox ~]$ comm file1.txt file2.txt
a
                b
                c
                d
        e
```

从以上结果可以看出，comm 输出了三列内容。第一列显示的是第一个文件中独有的行，第二列显示的是第二个参数文件中独有的行，第三列显示的则是两个文件所共有的行。comm 还支持-n 形式的参数选项，此处的 n 可以是 1、2 或者 3，使用时，它表示省略第 n 列的内容。例如，如果只想显示两个文件的共有行，便可以省略第 1 列和第 2 列的内容，示例如下。

```
[me@linuxbox ~]$ comm -12 file1.txt file2.txt
b
c
d
```

20.4.2 diff——逐行比较文件

与 comm 命令类似，diff 用于检测文件之间的不同。然而，diff 比 comm 更复杂，它支持多种输出形式，并且具备一次性处理大文件集的能力。diff 通常被软件开发者用于检查不同版本的源代码之间的差异，因为它能够递归检查源代码目录（通常称为源树）。diff 的常见用法就是创建 diff 文件和补丁，它们可以为诸如 patch（后面将会深入讨论）这样的命令所用，从而实现一个版本的文件更新为另一个版本。

将 diff 运用于前面例子中使用到的文件，我们会发现其默认形式的输出结果其实是对两者文件差异的一个简洁描述。

```
[me@linuxbox ~]$ diff file1.txt file2.txt
1d0
< a
4a4
> e
```

默认形式中，每一组改动的前面都有一个以“范围　执行操作　范围”形式（*range operation range*）表示的改变操作命令（见表 20-4），该命令会告诉程序对第一个文件的某个位置进行某种改变，便可实现与第二个文件内容一致。

表 20-4　diff 改变命令

改变操作	功能描述
r1ar2	将第二个文件中 r2 位置的行添加到第一个文件中的位置 r1 处
r1cr2	用第二个文件 r2 处的行替代第一个文件 r1 处的行
r1dr2	删除第一个文件 r1 处的行，并且删除的内容作为第二个文件 r2 行范围的内容

此格式中，范围 range 一般是由冒号隔开的起始行和末尾行组成。虽然，此格式是默认的（大多数情况下是为了兼容 POSIX 的同时向后兼容传统 UNIX 版本的 diff），但它并没有其他格式的应用广泛，上下文格式和统一格式才是比较普遍使用的格式。

上下文格式的输出结果如下。

```
[me@linuxbox ~]$ diff -c file1.txt file2.txt
*** file1.txt    2012-12-23 06:40:13.000000000 -0500
--- file2.txt    2012-12-23 06:40:34.000000000 -0500
***************
*** 1,4 ****
- a
  b
  c
  d
--- 1,4 ----
  b


  c
  d
+ e
```

该结果以两个文件的名字和时间信息开头，第一个文件用星号表示，第二个文件用破折号表示。输出结果的其余部分出现的星号和破折号则分别表示各自所代表的文件。其他的内容便是两个文件之间的差异组，包括文本的默认行号。第一组差异，以*** 1,4 ****开头，表示第一个文件中的第 1 行至第 4 行；第二组便以--- 1,4 ----开头 ，表示第二个文件的第 1 行至第 4 行。每个差异组的行都是以如下四个标识符之一开头，见表 20-5。

表 20-5　diff 上下文格式差异标识符

标示符	含义
（无）	该行表示上下文文本。表示两个文件共有的行
-	缺少的行。指此行内容只在第一个文件中出现，第二个文件中则没有
+	多余的行。此行内容只有第二个文件才有，第一个文件则没有
!	改变的行。两个版本的行内容都会显示出来，每一个都各自出现在差异组中相应的部分

统一格式与上下文格式相似但是更简明，此格式用-u 选项指定。

```
[me@linuxbox ~]$ diff -u file1.txt file2.txt
--- file1.txt    2012-12-23 06:40:13.000000000 -0500
+++ file2.txt    2012-12-23 06:40:34.000000000 -0500
@@ -1,4 +1,4 @@
-a
 b
 c
 d
+e
```

上下文格式和统一格式之间最显著的区别就是，统一格式下没有重复的文本行，这使得统一格式的输出结果比上下文格式的精简。上例中，也输出了上下文格式中出现的文件时间信息，并且后面紧跟着“@@ -1,4 +1,4 @@”字符串，它表示差异组中描述的两个文件各自的行范围。此字符串之后便是行本身，其中包含默认的三行文本内容。正如表 20-6 所示，每一行都以下面的三个可能的标示符开头。

表 20-6 diff 统一格式的差异标示符

字符	含义
（无）	两个文件共有的行
-	相对于第二个文件而言，第一个文件中没有的行
+	第一个文件多余的行

20.4.3 patch——对原文件进行 diff 操作

patch 命令用于更新文本文件。它利用 diff 命令的输出结果将较旧版本的文件升级成较新版本。下面看一个众所周知的例子，Linux 内核是由一个很大的、组织松散的志愿者团队开发的，其源代码处于持续不断更新中。Linux 内核包含几百万行代码，所以相比而言，某位开发成员每次所做的改变是如此微不足道。因此，对于每位开发者来说，每对代码改动一次就得向其他开发者发送整个内核源代码树是多么不切实际。事实上，一般只要发送 diff 补丁文件即可，diff 补丁的内容是内核从较旧版本转变为较新版本所经历的改变，接收者然后使用 patch 命令将这些改变应用于其自身的源代码树。diff/patch 有两个重要的优点。

- 与源代码树的大小相比，diff 文件很小。
- diff 文件非常简洁地描述了文件所做的改变，便于补丁的接收者快速对其进行评价。

当然，diff/patch 不仅仅局限于源代码，它适用于任何文本文件。因此，它同样适用于配置文件以及其他文本文件。

生成供 patch 使用的 diff 文件，GNU 文件系统建议采用如下方式使用 diff。

```
diff -Naur old_file new_file > diff_file
```

此处 *old_file* 和 *new_file* 既可以是单独的文件也可以是包含文件的目录，使用-r 参数选项则是为了进行递归目录树搜索。

一旦创建了 diff 文件，便可以将其用于修补原文件 *old_file*，从而升级为新文件 *new_file*：

```
patch < diff_file
```

以前面的测试文件为例：

```
[me@linuxbox ~]$ diff -Naur file1.txt file2.txt > patchfile.txt
[me@linuxbox ~]$ patch < patchfile.txt
patching file file1.txt
[me@linuxbox ~]$ cat file1.txt
b
c
d
e
```

本例中，我们创建了一个叫做 *patchfile.txt* 的 diff 文件，然后使用 patch 命令进行修补。请注意该命令行中，并没有给 patch 命令指定目标文件，因为 diff 文件（统一形式）已经在页眉中包含了文件名。可以发现，进行修补后，*file1.txt* 便与 *file2.txt* 一致了。

patch 有很多参数选项，而且有很多额外的工具命令可以分析并编辑这些补丁文件。

20.5 非交互式文本编辑

之前所描述的文本编辑大多都是交互式的，也就是说只要手动移动鼠标然后输入需要进行的改变，然而，也可以用非交互的方式进行文本编辑。例如，可以只用一个简单的命令一次性更改多个文件。

20.5.1 tr——替换或删除字符

tr 是替换字符命令，可以将其看作一种基于字符的查找和替换操作。所谓替换，实际是指将字符从一个字母更换为其他字母。例如，将小写字母转变为

大写字母。如下便是一个利用 tr 进行大小写字母替换的例子。

```
[me@linuxbox ~]$ echo "lowercase letters" | tr a-z A-Z
LOWERCASE LETTERS
```

该输出结果表明，tr 可对标准输入进行操作并且将结果以标准形式输出。tr 有两个参数：等待转换的字符集和与之相对应的替换字符集。字符集的表示方法可以是下面三种方式中的任一种。

- 枚举列表；例如，ABCDEFGHIJKLMNOPQRSTUVWXYZ。
- 字符范围；例如，A-Z。请注意，这种方法有时会与其他命令一样受限于同一个问题（由于不同系统的排序顺序），因此使用时要小心。
- POSIX 字符类；例如，[:upper:]。

多数情况下，这两个字符集应该是同等长度；然而，第一个字符集比第二个字符集长也是可能的，如下便是一个将多个字符替换为单个字符的例子。

```
[me@linuxbox ~]$ echo "lowercase letters" | tr [:lower:] A
AAAAAAAAA AAAAAAA
```

除了替换，tr 还可以直接从输入流中删除字符。本章前篇，讨论了将 MS-DOS 类型的文本文件向 UNIX 类型转换的问题。要进行这样的转换，需要移除每行末尾的回车符，如此便可以用 tr 命令解决，如下所示。

```
tr -d '\r' < dos_file > unix_file
```

这里的 dos_file 是待转换的文件，而 UNIX_file 则是转换的结果文件。此命令形式使用了转义字符\r 代替回车符。如若想了解 tr 支持的所有转义符号和字符类，请查看 tr 的帮助手册。

```
[me@linuxbox ~]$ tr —help
```

ROT13：保密性不强的编码环

tr 的一个有趣用法就是可以进行 ROT13 文本编码。ROT13 是一种基于简单替换算法的加密方式。ROT13 这个名字有点抽象，“文本模糊”应该更具体一点。在文本中应用该加密方式，有时是为了隐藏潜在的攻击性内容。该方法仅仅是将每个字母在字母表中向上移动了 13 位，由于正好移动了字母表中字母总数 26 的一半，所以再执行一次此算法文本便可以恢复原样。示例如下，使用 tr 进行编码。

```
echo "secret text" | tr a-zA-Z n-za-mN-ZA-M
frperg grkg
```

再进行一次同样的步骤即得到了译码如果：

```
echo "frperg grkg" | tr a-zA-Z n-za-mN-ZA-M
secret text
```

许多 email 程序以及 Usenet 新闻阅读器都支持 ROT13 编码。维基百科上有一篇关于该主题的很好的文章，大家有兴趣的可以看一看，网址为：*http://en.wikipedia.org/wiki/ROT13*。

tr 还有另外一个奇妙的用法。使用-s 选项，tr 可以“挤兑”（删除）重复出现的字符，示例如下。

```
[me@linuxbox ~]$ echo "aaabbbccc" | tr -s ab
abccc
```

本例字符串中含有重复字符，通过给 tr 指定 ab 字符集，便消除了该字符集中重复的“a”和“b”字母，而字母 c 并没有改变，因为 tr 设定的字符集中并没有包含 c。请注意重复的字符必须是毗邻的，否则该挤兑操作将不起作用。

```
[me@linuxbox ~]$ echo "abcabcabc" | tr -s ab
abcabcabc
```

20.5.2 sed——用于文本过滤和转换的流编辑器

sed 是 *stream editor*（流式编辑器）的缩写，它可以对文本流、指定文件集或标准输入进行文本编辑。sed 功能非常强大，并且从某种程度上来说，它还是一个比较复杂的命令（有整本书的内容对它进行了讲解），所以本书无法涉及其所有用法。

sed 的用法，总得来说，首先给定 sed 某个简单的编辑命令（在文本行中）或是包含多个命令的脚本文件名，然后 sed 便对文本流的内容执行给定的编辑命令。下面是一个非常简单 sed 应用实例。

```
[me@linuxbox ~]$ echo "front" | sed 's/front/back/'
back
```

该例先利用 echo 生成了只包含一个单词的文本流，然后将该文本流交给 sed 处理，而 sed 则对文本流执行 s/front/back/指令，最后输出“back”作为运行结果。所以可以认为，本例中的 sed 与 vi 中的替代（查找与替换）命令相似。

sed 中的命令总是以单个字母开头。上例中，替换命令便由字母 s 代替，其后紧跟替换字符，替换字符由作为分界符的斜线字符分开。分界符的选择是随

意的，习惯上一般使用斜线，但是 sed 支持任意字符作为分界符，sed 会默认紧跟在 sed 的命令之后的字符为分界符。下面的命令行能得到相同的效果。

```
[me@linuxbox ~]$ echo "front" | sed 's_front_back_'
back
```

由于下划线是紧跟在命令字符“s”之后的，所以它便是分界符。由此可见，这种自动设定分界符的能力增强了命令行的可读性。

sed 中的多数命令允许在其前添加一个地址，该地址用来指定输入流的哪一行被编辑。如果该地址省略了，便会默认对输入流的每一行执行该编辑命令。最简单的地址形式就是一个行号。例如，在上例中增加一个“1”，如下所示。

```
[me@linuxbox ~]$ echo "front" | sed '1s/front/back/'
back
```

增加的“1”表示此替换操作只对输入流的第一行起作用。当然也可以指定其他行号，如下所示。

```
[me@linuxbox ~]$ echo "front" | sed '2s/front/back/'
front
```

结果显示此替换操作并未执行，这是因为该输入流中并没有第二行。地址可以有多种方式表达，表 20-7 列出了最常用的几个。

表 20-7 sed 的地址表达法

地址	功能说明
N	n 是正整数表示行号
$	最后一行
/regexp/	用 POSIX 基本正则表达式描述的行。请注意，这里的正则表达式用的是斜线作为分界符，当然，也可以自己选择分界符，只要用\cregexpc 选项指定即可，这里的 c 就是用于取代斜杠的分界符
addr1,addr2	行范围，表示从 addr1 至 addr2 的所有行。地址可以是上面所述形式的任何一种
first~step	代表行号从 first 行开始，以 step 为间隔的所有行。例如，1~2 是指所有奇数行，而 5~5 是指第 5 行和以及随后的所有是 5 的倍数的行
addr1,+n	addr1 行及其之后的 n 行
addr!	出了 addr 行之外的所有行，addr 可以用上面的任何一种形式表达

接下来，我们会用本章前面所使用的 *distros.txt* 文件演示多种不同形式的地址表达。首先，用行号范围的表达方式如下所示。

```
[me@linuxbox ~]$ sed -n '1,5p' distros.txt
SUSE            10.2     12/07/2006
```

```
Fedora          10      11/25/2008
SUSE            11.0    06/19/2008
Ubuntu          8.04    04/24/2008
Fedora          8       11/08/2007
```

此例显示了 *distros.txt* 文件中第 1 行到第 5 行的内容，利用了 p 命令输出指定匹配行的内容，从而完成上述操作。然而，要想得到正确结果，就必须添加选项-n（不会自动打印选项）以防 sed 会默认输出每一行的内容。

下面尝试使用正则表达式。

```
[me@linuxbox ~]$ sed -n '/SUSE/p' distros.txt
SUSE            10.2    12/07/2006
SUSE            11.0    06/19/2008
SUSE            10.3    10/04/2007
SUSE            10.1    05/11/2006
```

此处用的是以斜杠隔开的正则表达式/SUSE/，查找包含 SUSE 字符串的文本行，该用法与 grep 的用法类似。

最后，尝试在地址前添加表示否定意义的感叹号（!），用法如下。

```
[me@linuxbox ~]$ sed -n '/SUSE/!p' distros.txt
Fedora          10      11/25/2008
Ubuntu          8.04    04/24/2008
Fedora          8       11/08/2007
Ubuntu          6.10    10/26/2006
Fedora          7       05/31/2007
Ubuntu          7.10    10/18/2007
Ubuntu          7.04    04/19/2007
Fedora          6       10/24/2006
Fedora          9       05/13/2008
Ubuntu          6.06    06/01/2006
Ubuntu          8.10    10/30/2008
Fedora          5       03/20/2006
```

这样我们得到了理想输出结果，即除了那些与正则表达式匹配的行，其他所有行都显示出来了。

到目前为止，我们已经介绍了 sed 的两个编辑命令——s 和 p，表 20-8 是一张更完整的基本编辑指令表。

表 20-8　sed 基本编辑指令

命令	功能描述
=	输出当前行号
a	在当前行后附加文本
d	删除当前行
i	在当前行前输入文本

续表

命令	功能描述
p	打印当前行。默认情况下，sed 会输出每一行并且只编辑文件内那些匹配指定地址的行。当指定-n 选项时，默认操作会被覆盖
q	退出 sed 不再处理其他行。如果没有指定-n 选项，就会输出当前行
Q	直接退出 sed 不再处理行
s/regexp/replacement/	将 regexp 的内容替换为 replacement 代表的内容。replacement 可能会包含特殊字符&，它代表的其实就是 regexp 所表示内容。除此之外，replacement 也可能包含\1 到\9 的序列，它们代表的是 regexp 中相应位置的描述内容。学习接下来关于回参考的讨论可以了解更多这方面的知识。跟在 replacement 后面的反斜杠，可以指定一个可选择的标志以修改 s 命令的行为
y/set1/set2	将字符集 set1 转换为字符集 set2。请注意，与 tr 不同，sed 要求这两个字符集等长

s 命令是目前为止使用最普遍的编辑命令。接下来，我们通过编辑 *distros.txt* 文件来演示其强大功能的一小部分。之前已经讨论过 *distros.txt* 文件中的时间字段并不是以“计算机友好”的形式存储，因为此时间形式是 MM/DD/YYYY，YYYY-MM-DD 这样的形式会方便很多（更容易排序）。但如果手动更改文件，不仅浪费时间而且容易出错，而 sed 可以一步完成这样的操作。

```
[me@linuxbox ~]$ sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1
-\2/' distros.txt
SUSE            10.2     2006-12-07
Fedora          10       2008-11-25
SUSE            11.0     2008-06-19
Ubuntu          8.04     2008-04-24
Fedora          8        2007-11-08
SUSE            10.3     2007-10-04
Ubuntu          6.10     2006-10-26
Fedora          7        2007-05-31
Ubuntu          7.10     2007-10-18
Ubuntu          7.04     2007-04-19
SUSE            10.1     2006-05-11
Fedora          6        2006-10-24
Fedora          9        2008-05-13
Ubuntu          6.06     2006-06-01
Ubuntu          8.10     2008-10-30
Fedora          5        2006-03-20
```

哇塞！这个命令行看起来还真复杂，但它确实有效。只需一步，就改变了文件中的日期形式。此例充分解释了为什么正则表达式有时被开玩笑的称为“只写”介质。因为我们可以编写正则表达式，但有时却读不懂它们。在被该命令行吓跑之前，这我们还是先来分析它的各组成部分的含义。首先，sed 命令有其基本结构，如下所示。

```
sed 's/regexp/replacement/' distros.txt
```

接下来我们需要理解将日期分隔开来的正则表达式。由于它是以 MM/DD/YYYY 的形式存在，并且出现在行末尾，所以使用如下的表达式。

```
[0-9]{2}/[0-9]{2}/[0-9]{4}$
```

该表达式的匹配格式：两位数字、斜杠、两位数字、斜杠、4 位数字以及行尾标志。所以这代表了 *regexp* 表达式的形式，但是怎么处理 *replacement* 的表达式？解决这一问题，我们必须引进正则表达式的一个新特性，该特性一般存在于那些使用 BRE 的应用中。此特性称为回参考，并且工作方式类似于此，即如果 *replacement* 中出现了*n* 转义字符，并且这里的 *n* 是 1-9 之间的任意数字，那么此转义字符就是指前面正则表达式中与之相对应的子表达式。如何创建该表达式，可以简单地将其括于括号中，如下所示。

```
([0-9]{2})/([0-9]{2})/([0-9]{4})$
```

现在，我们便有了三个子表达式。第一个的内容是月份，第二个内容是具体的日，第三个内容则是指年份。于是便可用如下命令行构建替换字符。

```
\3-\1-\2
```

该表达式表示的顺序如下：年份、斜杠、月份、斜杠和具体的日。

于是，整个命令行如下。

```
sed 's/([0-9]{2})/([0-9]{2})/([0-9]{4})$/\3-\1-\2/' distros.txt
```

但是仍有两个遗留问题：第一，当 sed 试图编译 s 命令时，正则表达式中多余的斜杠会令 sed 混淆；第二，sed 默认情况下只接受基本正则表达式，所以正则表达式中的部分元字符会被当成文字字符。我们可以利用反斜杠来避免这些冒犯字符，从而一次性解决这两个问题。

```
sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/' dis
tros.txt
```

这样便大功告成了！

s 命令的另外一个特点，就是替换字符串后面可以紧跟可选择标志符。其中最重要的标志符就是 g，该标志告诉 sed 对每行的所有匹配项进行替换操作，而不是默认的只替换第一个匹配项。

示例如下。

```
[me@linuxbox ~]$ echo "aaabbbccc" | sed 's/b/B/'
aaaBbbccc
```

我们可以看到执行了替换操作，但只是对第一个字母 b 有效，剩下的字母并没有改变。通过增加 g 标志符，便可以对所有的 b 进行替换操作。

```
[me@linuxbox ~]$ echo "aaabbbccc" | sed 's/b/B/g'
aaaBBBccc
```

到目前为止，我们只使用了命令行方式向 sed 传送操作命令，其实也可以用-f 选项建立更复杂的命令脚本文件。作为演示实例，我们运用 *distros.txt* 文件结合 sed 创建一个报告。该报告的构成有顶端标题、修改时间和大写字母组成的所有发行版本名。进行这些操作之前，我们需要编写一个脚本文件，所以启动文本编辑器并输入如下内容。

```
# sed script to produce Linux distributions report

1 i\
\
Linux Distributions Report\

s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

将此 sed 脚本保存为 *distros.sed*，并且照如下方式运行。

```
[me@linuxbox ~]$ sed -f distros.sed distros.txt

Linux Distributions Report

SUSE            10.2    2006-12-07
FEDORA          10      2008-11-25
SUSE            11.0    2008-06-19
UBUNTU          8.04    2008-04-24
FEDORA          8       2007-11-08
SUSE            10.3    2007-10-04
UBUNTU          6.10    2006-10-26
FEDORA          7       2007-05-31
UBUNTU          7.10    2007-10-18
UBUNTU          7.04    2007-04-19
SUSE            10.1    2006-05-11
FEDORA          6       2006-10-24
FEDORA          9       2008-05-13
UBUNTU          6.06    2006-06-01
UBUNTU          8.10    2008-10-30
FEDORA          5       2006-03-20
```

正如读者所看到的，输出显示了理想结果，但是它到底是如何工作的呢？让我们再次查看脚本文件，并使用 cat 将行号都标注出来。

```
[me@linuxbox ~]$ cat -n distros.sed
   1  # sed script to produce Linux distributions report
   2
   3  1 i\
   4  \
   5  Linux Distributions Report\
   6
   7  s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
   8  y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

第一行只是一个声明。与 Linux 系统中的许多配置文件和编程语言一样，声明一般都是以“#”符号开头。剩下的则是一些人类可理解的文本。声明可以放于脚本文件的任何位置（只要不在命令行中），并且对于任何一个需要验证或是维护该脚本的人都很有用。

第二行是空白行。与声明一样，空白行也是为了增加可读性。

多数 sed 命令都支持行地址，这些行地址用于指定哪些输入行执行指定操作。行地址可以用简单的行号来描述，也可以用行号范围以及“$”表示，“$”是一个表示文本最后一行的特殊符号。

第 3~6 行包含的则是要插入文本中第一行的内容。i 命令后面紧跟转义回车符，转义回车符由反斜杠和回车符组成，亦称为行继续符。此种先反斜杠后回车符的顺序，可以用于包括 shell 脚本在内的许多场合，可确保文本流中嵌入回车符但不会告诉编译器（在 sed 这个例子中）已经到了行末尾。i 命令、a 命令（追加文本）以及 c 命令（替换文本）能作用于多个文本行，只要除了最后一行的每行都以行继续符结尾。该脚本文件的第 6 行确实是所输入文本的末尾行，并且标志 i 命令结尾的符号是一个简单的回车符而不再是行继续符。

注意 行继续符是由反斜杠后紧跟回车符组成，两者之间不容许有任何空格。

第 7 行是查找和替换命令。由于该命令前并未指定地址，所以该命令将对输入流的每一行进行操作。

第 8 行则实现了小写字母向大写字母的转变。注意，与 tr 不同，sed 中的 y 命令并不支持字符范围（例如[a-z]），也不支持 POSIX 字符类。同样，由于 y 命令前并没有指定地址，所以将对输入流的每一行执行操作。

偏爱 sed 的人同样会喜欢……

sed 是一个功能非常强大的程序，它可以对文本流执行非常复杂的编辑任务。它通常用于简单的单命令行任务而不是长脚本。对于更大些的任务，许多用户则偏向于其他工具。其中最受欢迎的有 awk 和 perl。这两个命令已经超出了本书所谈论的命令范畴，并且延伸到了完整的编程语言领域。尤其是 perl，通常取代 shell 脚本用于执行系统管理和管理员任务，同时也是一个非常受欢迎的 web 开发介质。而 awk 则更专业化，数据处理是其强项。它与 sed

的相似之处就是 awk 命令通常也是逐行处理文本文件，并且使用方法也继承了 sed 的地址后面加上执行操作这一个概念。虽然 awk 和 perl 都不在本书的讨论范围中，但对于 Linux 命令行用户来说它们仍然是非常有用的工具。

20.5.3 aspell——交互式拼写检查工具

最后一个所要讨论的文本工具就是 aspell，它是交互式的拼写检查工具。aspell 命令继承的是早期的 ispell 命令，并且多数情况下，可以直接取代 ispell。虽然 aspell 命令通常为那些需要进行拼写检查的程序所用，但它同样可以作为一个独立于命令行的工具发挥其效用。它可以智能地检查不同类型文本文件的错误，包括 HTML 文件、C/C++程序、email 消息以及其他专业的文本文件。

检查一篇简单散文的拼写错误，可以用如下方式使用 aspell。

```
aspell check textfile
```

此处的 testfile 是要进行检查的文件名。作为实例进行讲解，下面创建了一个简单的 foo.txt 文本文件，它包含一些故意的拼写错误。

```
[me@linuxbox ~]$ cat > foo.txt
The quick brown fox jimped over the laxy dog.
```

接下来使用 aspell 检查文件中的拼写错误。

```
[me@linuxbox ~]$ aspell check foo.txt
```

由于 aspell 在检验模式下是与用户交互的，所以我们会看到如下的显示界面。

```
The quick brown fox jimped over the laxy dog.

1) jumped                              6) wimped
2) gimped                              7) camped
3) comped                              8) humped
4) limped                              9) impede
5) pimped                              0) umped
i) Ignore                              I) Ignore all
r) Replace                             R) Replace all
a) Add                                 l) Add Lower
b) Abort                               x) Exit

?
```

显示内容的顶部，被怀疑错误的字符是以高亮的形式显示的。中间部分，有 10 个标号从 0～9 的替换拼写建议以及其他可能的动作选项。最后，末端有一个提示框供用户进行操作选择。

假定我们输入 1，则 aspell 会将错误的单词用 jumped 取代并且继续处理下一个错误单词 laxy。如果选择替代单词 lazy，aspell 便执行此替换操作然后终止程序。aspell 命令检查结束后，可以再次查看文件，会发现那些拼写错误的单词已经改正过来，如下所示。

```
[me@linuxbox ~]$ cat foo.txt
The quick brown fox jumped over the lazy dog.
```

除非额外指定了命令行选项--dont-backup，不然 aspell 将会创建一个包含原文本内容的备份文件，此备份文件文件名则由原文件名加上后缀*.bak* 组成。

sed 其实有更强大的编辑功能，恢复 *foo.txt* 文件中原有的拼写错误以便再次利用：

```
[me@linuxbox ~]$ sed -i 's/lazy/laxy/; s/jumped/jimped/' foo.txt
```

sed 选项-i 告诉 sed“原地”编辑文件，这表示 sed 不会将编辑结果送至标准输出，而是将改变后的文本重新写入文件中。同样可以看出，一个命令行中可以输入多个编辑命令，只要用分号将它们隔开即可。

接下来就来讨论一下 aspell 如何处理不同类型的文件。使用诸如 vim 的文本编辑器（可以挑战性地尝试使用 sed），给文件增加一些 HTML 语言，如下所示。

```
<html>
        <head>
                <title>Mispelled HTML file</title>
        </head>
        <body>
                <p>The quick brown fox jimped over the laxy dog.</p>
        </body>
</html>
```

现在，如果试图检查修改后该文件的拼写错误，便会遇到一个问题。照如下方式输入命令行。

```
[me@linuxbox ~]$ aspell check foo.txt
```

便会得到如下输出结果。

```
<html>
        <head>
                <title>Mispelled HTML file</title>
        </head>
        <body>
                <p>The quick brown fox jimped over the laxy dog.</p>
        </body>
</html>

1) HTML                                4) Hamel
```

```
2) ht ml                                5) Hamil
3) ht-ml                                6) hotel
i) Ignore                               I) Ignore all
r) Replace                              R) Replace all
a) Add                                  l) Add Lower
b) Abort                                x) Exit

?
```

aspell 会认为 HTML 标签的所有内容是拼写错误。该问题可以通过增加-H（HTML）模式选项克服，示例如下。

```
[me@linuxbox ~]$ aspell -H check foo.txt
```

可以得到如下结果：

```
<html>
      <head>
            <title>Mispelled HTML file</title>
      </head>
      <body>
            <p>The quick brown fox jimped over the laxy dog.</p>
      </body>
</html>

1) Mi spelled                           6) Misapplied
2) Mi-spelled                           7) Miscalled
3) Misspelled                           8) Respelled
4) Dispelled                            9) Misspell
5) Spelled                              0) Misled
i) Ignore                               I) Ignore all
r) Replace                              R) Replace all
a) Add                                  l) Add Lower
b) Abort                                x) Exit

?
```

使用-H 选项后，HTML 语言部分就被忽略了，只有那些非 HTML 标签部分才会被检查。这种模式下，HTML 标签内容被忽略了并且不会进行拼写检查。然而，Alt 标签的内容，在这种模式下则是要检查的。

注意 默认情况下，aspell 会忽略文本中的 URL 和 email 地址，该行为可以通过设置命令行选项覆盖。同样也可以指定哪些标签内容需要被检查，而哪些可跳过检查。具体内容，可以查看 aspell 的 man 手册页。

20.6 本章结尾语

本章主要介绍了一些用于文本编辑的命令行工具，下一章会介绍更多的这

类工具。不得不承认，虽然文中已经列举了其中部分命令的应用实例，但你仍然可能会对如何使用以及为什么使用这些工具存在疑问，而且答案似乎并不是那么显而易见。在后面的章节中，相信大家会逐渐发现这些工具其实是解决一组实际问题的基本工具，在后面接触到 shell 脚本的知识时，这些工具才会真正体现其价值。

20.7　附加项

还有许多更有趣的文本操作命令值得探索。这些命令包括 split（将文件分成多个部分）、csplit（基于上下文将文件分块）以及 sdiff（左右并排显示文件差异并作比较）。

第21章

格式化输出

本章继续讨论与文本相关的工具，重点讲一些用于格式化文本输出而非改变文本自身内容的命令。这些命令通常用于文本的打印，而“打印”这一主题将在下一章介绍。本章将要讨论的命令如下所示。

- nl：对行进行标号。
- fold：设定文本行长度。
- fmt：简单的文本格式化工具。
- pr：格式化打印文本。
- printf：格式化并打印数据。
- groff：文档格式化系统。

21.1 简单的格式化工具

首先让我们看一些简单的格式化工具，它们多数都是“单目的”程序，而且一般执行一些不复杂的操作，它们一般用于一些小的任务，并作为管道传输和脚本的一部分。

21.1.1 nl——对行进行标号

nl 命令是一个非常神秘的工具，用于完成一个非常简单的任务：对行进行编号。就其最简单用法，与 cat –n 很相似。

```
[me@linuxbox ~]$ nl distros.txt | head
     1 SUSE              10.2     12/07/2006
     2 Fedora            10       11/25/2008
     3 SUSE              11.0     06/19/2008
     4 Ubuntu            8.04     04/24/2008
     5 Fedora            8        11/08/2007
     6 SUSE              10.3     10/04/2007
     7 Ubuntu            6.10     10/26/2006
     8 Fedora            7        05/31/2007
     9 Ubuntu            7.10     10/18/2007
    10 Ubuntu            7.04     04/19/2007
```

和 cat 命令一样，nl 既支持多个文件名作为命令行参数，也支持标准输入。然而，nl 可以进行多种复杂的编号，因为它有许多参数选项，且支持原始形式的标记。

nl 进行标号时支持一个叫做逻辑页的概念，所以它可以重置（重新开始）数值序列。通过合理运用参数选项，nl 可以设置起始编号为特定的值，并在有限的范围内设置其格式。逻辑页可以进一步分解为逻辑页标题、正文和页脚。在每一个部分中，行号都可以重置并/或分配不同的风格。如果 nl 的输入参数是多个文件，那么 nl 会把它们当作一个文本流整体。文本流中的每一个部分都由一些看起来非常奇怪的标记来区别，表 21-1 列出了部分标记。

表 21-1 nl 标记

标记	含义
\:\:\:	逻辑页页眉开头
\:\:	逻辑页正文开头
\:	逻辑页页脚开头

表 21-1 中的每一个标记元素在一行中只允许出现一次。每次处理完一个标

记元素后，nl 便将其从文本流中删除。

表 21-2 列出了 nl 的常用选项。

表 21-2　常用的 nl 选项

选项	含义
-b *style*	按照 *style* 格式对正文进行编号，这里的 *style* 是下面类型中的一个 • **a**　对每行编号 • **t**　仅仅对非空白行编号，此选项是默认的 • **n**　不对任何行进行编号 • **pregexp**　只对与基本正则表达式匹配的行进行编号
-f *style*	以 *style* 的格式对页脚进行编号。默认选项是 n（无）
-h *style*	以 *style* 的格式对标题进行编号。默认选项是 n（无）
-i *number*	设置页编号的步进值为 *number*。默认值为 1
-n *format*	设置编号格式为 *format*，此处的 *format* 可以是如下表示中的一种 • **ln**　左对齐，无缩进 • **rn**　右对齐，无缩进。这是默认选项 • **rz**　右对齐，有缩进
-p	在每个逻辑页的开始不再进行页编码重置
-s *string*	在每行行号后面增加 *string* 作为分隔符。默认的情况下是一个简单的 tab 制表符
-v *number*	将每个逻辑页的第一个行号设为 *number*。默认是 1
-w *width*	设置行号字段的宽度为 *width*。默认值是 6

不得不承认，用户可能并不会那么频繁地进行行编号，但是，用户可以利用 nl 结合其他工具进行更复杂的任务。基于上一章的基础，我们生成一个 Linux 发行版本的报告。由于我们会使用 nl，报告中最好包含标题/正文/页脚等标记。我们可以用上章提到的 sed 脚本添加这些标记，用文本编辑器对 sed 脚本文件做如下改动，并将其存于 *distros-nl.sed* 文件中。

```
# sed script to produce Linux distributions report

1 i\
\\:\\:\\:\
\
Linux Distributions Report\
\
Name                Ver.      Released\
----                ----      --------\
\\:\\:
s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
$ a\
\\:\
\
End Of Report
```

该脚本文件完成了向原报告中插入 nl 逻辑页标记以及在报告末尾添加页脚内容的任务。请注意，输入标记时必须使用双反斜杠，否则 sed 会将它们解释为转义字符。

接下来，我们便可以结合 sort、sed 和 nl 创建一个增强版的报告。

```
[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-nl.sed | nl


       Linux Distributions Report

        Name    Ver.    Released
        ----    ----    --------

     1  Fedora  5       2006-03-20
     2  Fedora  6       2006-10-24
     3  Fedora  7       2007-05-31
     4  Fedora  8       2007-11-08
     5  Fedora  9       2008-05-13
     6  Fedora  10      2008-11-25
     7  SUSE    10.1    2006-05-11
     8  SUSE    10.2    2006-12-07
     9  SUSE    10.3    2007-10-04
    10  SUSE    11.0    2008-06-19
    11  Ubuntu  6.06    2006-06-01
    12  Ubuntu  6.10    2006-10-26
    13  Ubuntu  7.04    2007-04-19
    14  Ubuntu  7.10    2007-10-18
    15  Ubuntu  8.04    2008-04-24
    16  Ubuntu  8.10    2008-10-30



        End Of Report
```

多个命令进行管道传输得到了需求结果。首先，我们根据发行版名称和发行时间（字段 1 和字段 2）进行排序；然后用 sed 处理排序结果，添加了报告的页眉（包括 nl 的逻辑页标记）和页脚；最后，我们执行了 nl 命令。nl 在默认情况下，只会对属于逻辑页正文部分的文本流进行行编号。

我们可以重复执行 nl，并尝试不同的参数选项。如下所示列举了一些有趣的参数。

```
nl -n rz
```

以及

```
nl -w 3 -s ' '
```

21.1.2 fold——将文本中的行长度设定为指定长度

fold 是一个将文本行以指定长度分解的操作。与其他命令类似，fold 支持一

个或多个文本文件或是标准输入作为输入参数。向 fold 输入一个简单的文本流，便可了解其工作方式。

```
[me@linuxbox ~]$ echo "The quick brown fox jumped over the lazy dog." | fold
-w 12
The quick br
own fox jump
ed over the
lazy dog.
```

这样，我们便能明白 fold 到底完成了什么操作。echo 命令输出的文本被指定了-w 选项的 fold 分解成了片段。本例中，指定了行的宽度为 12 个字符。如果没有指定行宽，则默认是 80 个字符宽。请注意，fold 在断行时并不会考虑单词边界。而增加-s 选项，可使 fold 在到达 width 字符数前的最后一个有效空格处将原文本行断开，示例如下。

```
[me@linuxbox ~]$ echo "The quick brown fox jumped over the lazy dog." | fold
-w 12 -s
The quick
brown fox
jumped over
the lazy
dog.
```

21.1.3 fmt——简单的文本格式化工具

fmt 命令同样会折叠文本，另外还包括更多其他功能。它既可以处理文件也可以处理标准输入，并对文本流进行段落格式化。就其基本功能而言，它可以在保留空白行和缩进的同时对文本行进行填充和连接。

作为演示的文本内容，不如从 fmt 的帮助手册页中复制一些内容吧。

```
  `fmt' reads from the specified FILE arguments (or standard input if none
are given), and writes to standard output.

  By default, blank lines, spaces between words, and indentation are
preserved in the output; successive input lines with different
indentation are not joined; tabs are expanded on input and introduced on
output.

  `fmt' prefers breaking lines at the end of a sentence, and tries to avoid
line breaks after the first word of a sentence or before the last word of a
sentence. A "sentence break" is defined as either the end of a paragraph or a
word ending in any of `.?!', followed by two spaces or end of line, ignoring
any intervening parentheses or quotes. Like TeX, `fmt' reads entire
"paragraphs" before choosing line breaks; the algorithm is a variant of that
given by Donald E. Knuth and Michael F. Plass in "Breaking Paragraphs Into
Lines", `Software--Practice & Experience' 11, 11 (November 1981), 1119-1184.
```

将这段文字复制到文本编辑器中，并将其存为 *fmt-info.txt* 文本文件。现在，

假定我们需要重新格式化该文本，以满足每行 50 个字符宽的规则。那么我们可以输入 fmt 结合-w 选项完成这样的格式化。

```
[me@linuxbox ~]$ fmt -w 50 fmt-info.txt | head
   `fmt' reads from the specified FILE arguments
   (or standard input if
none are given), and writes to standard output.

   By default, blank lines, spaces between words,
   and indentation are
preserved in the output; successive input lines
with different indentation are not joined; tabs
are expanded on input and introduced on output.
```

这个输出结果还真是奇怪。也许，实际上我们应该认真地阅读一遍下面的文字，因为它解释了事情发生的原委。

> 默认情况下，空白行、单词之间的空格和缩进都保留在输出结果中；不同缩进量的连续输入行并不进行拼接；制表符会在输入中扩展并直接输出。

所以，fmt 保留了第一行的缩进。幸运的是，fmt 提供了一个参数选项以修正这一问题。

```
[me@linuxbox ~]$ fmt -cw 50 fmt-info.txt
   `fmt' reads from the specified FILE arguments
(or standard input if none are given), and writes
to standard output.

   By default, blank lines, spaces between words,
and indentation are preserved in the output;
successive input lines with different indentation
are not joined; tabs are expanded on input and
introduced on output.

   `fmt' prefers breaking lines at the end of a
sentence, and tries to avoid line breaks after
the first word of a sentence or before the
last word of a sentence. A "sentence break"
is defined as either the end of a paragraph
or a word ending in any of `.?!', followed


by two spaces or end of line, ignoring any
intervening parentheses or quotes. Like TeX,
`fmt' reads entire "paragraphs" before choosing
line breaks; the algorithm is a variant of
that given by Donald E. Knuth and Michael F.
Plass in "Breaking Paragraphs Into Lines",
`Software--Practice & Experience' 11, 11
(November 1981), 1119-1184.
```

这样一来，输出结果看起来顺眼了很多。由此可见，通过增加-c 选项，我

们便得到了理想输出结果。

fmt 有一些有趣的选项，见表 21-3。

表 21-3　fmt 选项

选项	功能描述
-c	在“冠边缘”模式下运行。此模式保留段落前两行的缩进，随后的行都与第二行的缩进对齐
-p string	只格式化以前缀字符串 *string* 开头的行。格式化后，*string* 的内容仍然会作为每一个格式化行的前缀。该选项可以用于格式化内容是源代码的文本，例如，任何以“#”号作为声明开头的编程语言或是配置文件都可以通过指定-p ‘#’选项进行格式化，指定该选项后可保证只格式化声明中的内容。具体的可见后续例子
-s	“仅截断行”模式。在此模式下，将会只根据指定的列宽截断行。而短行并不会与其他行结合。此模式适用于格式化文本但不需要行拼接的场合，比如代码等
-u	字符间隔统一。采取传统的“打字机风格”模式格式化文本，这意味着字符之间间隔一个空格字符，句子之间间隔两个空格字符。该模式对于删除齐行非常有用，所谓齐行就是指文本行被强迫与左右边缘对齐
-w width	格式化文本使每行文本不超过 *width* 个字符，默认值是 75。请注意，fmt 格式化文本时往往由于要保持行平衡而使得行实际宽度比指定宽度略小

-p 选项尤为有趣，通过它，我们可以选择性地格式化文件内容，前提是要格式化的文本行都以相同的字符序列开头。许多编程语言都使用 hash 符号（#）作为注释的开始，因此可以用此选项只格式化注释文本。下面创建一个有注释的类似于程序的文本文件。

```
[me@linuxbox ~]$ cat > fmt-code.txt
# This file contains code with comments.

# This line is a comment.
# Followed by another comment line.
# And another.

This, on the other hand, is a line of code.
And another line of code.
And another.
```

我们的示例文本包含了以“#”（“#”号后紧跟一个空格）字符串开头的注释以及“代码”行（虽然并不是真正意义上的代码）。现在，我们使用 fmt 格式化该注释内容而不改动代码。

```
[me@linuxbox ~]$ fmt -w 50 -p '# ' fmt-code.txt
# This file contains code with comments.

# This line is a comment. Followed by another
# comment line. And another.

This, on the other hand, is a line of code.
```

```
And another line of code.
And another.
```

请注意，毗邻的注释行已经拼接起来，但是保留了空白行以及不是以指定前缀开头的文本行。

21.1.4 pr——格式化打印文本

Pr 命令用于给文本标页码。打印文本时，通常希望将输出内容分成几页，并且每页的顶部和底部都留出几行空白行，这些空白行可以用于插入页眉和页脚。

示例如下，该命令行将 *distros.txt* 文件格式化为一系列非常短的页（只显示了前两页）。

```
[me@linuxbox ~]$ pr -l 15 -w 65 distros.txt


2012-12-11 18:27                distros.txt                Page 1


SUSE            10.2        12/07/2006
Fedora          10          11/25/2008
SUSE            11.0        06/19/2008
Ubuntu          8.04        04/24/2008
Fedora          8           11/08/2007


2012-12-11 18:27                distros.txt                Page 2


SUSE            10.3        10/04/2007
Ubuntu          6.10        10/26/2006
Fedora          7           05/31/2007
Ubuntu          7.10        10/18/2007
Ubuntu          7.04        04/19/2007
```

上例中，结合了-l 选项（页长）以及-w 选项（页宽）定义了一“页”内容包含 15 行，每行包含 65 个字符。pr 对 *distros.txt* 文件的内容进行分页，页与页之间则用几行空白行隔开，并且创建了一个包含文件修改时间、文件名以及页码的默认页眉。pr 命令有很多选项用于控制页面布局，详见第 22 章。

21.1.5 printf——格式化并打印数据

与本章中涉及的其他命令不一样，printf 命令并不适用于管道传输（也就是说它不支持标准输入），而且在命令行应用中它也不常见（多应用于脚本文件）。

那么它到底为什么会这么重要？因为它有如此广泛的应用范围。

printf（短语 print formatted 的缩写）起初是为 C 语言开发的，后来许多编程语言也都实现了这一功能，也包括 shell 环境。事实上，在 bash 中，printf 是内置的。

printf 的用法如下。

```
printf "format" arguments
```

该命令行给出了一个包含格式说明的字符串，然后将该格式应用于 arguments 所代表的输入内容，最后格式化结果送至标准输出。如下就是一个简单例子。

```
[me@linuxbox ~]$ printf "I formatted the string: %s\n" foo
I formatted the string: foo
```

该格式化字符串可以包含文字文本（如“I formatted the string”）、转义字符（如\n，即换行符）以及以%开头的表示转换规格的字符序列。上例中，转换规格%s 用于格式化字符串 foo 并将其结果输出。再看下面一个例子。

```
[me@linuxbox ~]$ printf "I formatted '%s' as a string.\n" foo
I formatted 'foo' as a string.
```

以上可以看出，%s 所代表的转换规格被字符串 foo 取代。s 表示格式化字符串数据，其他类型的数据则用其他指定字符表示。表 21-4 列出了常用的数据类型。

表 21-4　常用 printf 数据类型指定符

指定符	说明
D	将一个数字格式化为有符号的十进制表示形式
F	格式化数字并以浮点数的格式输出
O	将一个整数格式化为八进制格式的整数
s	格式化字符串
x	将一个整数格式化为十六进制的数，并且在使用字母时，用小写字母 a~f 表示
X	与 x 类似，只是字母用大写字母表示
%	打印文字符号“%”（例如，指定“%%”）

下例中，利用字符串“380”演示了每个转换说明符的使用效果：

```
[me@linuxbox ~]$ printf "%d, %f, %o, %s, %x, %X\n" 380 380 380 380 380 380
380, 380.000000, 574, 380, 17c, 17C
```

本例指定了 6 个转换说明符，并向 printf 提供了 6 个输入参数。得到了 6 种转换说明符的输出效果。

转换说明符也可以通过增加一些可选组件以对输出效果进行调整。一个完整的转换规格可能会包含如下内容。

```
%[flags][width][.precision]conversion_specification
```

当使用多个可选组件时，这些组件必须按照上面的顺序才能被正确编译。表 21-5 给出了每个组件的说明。

表 21-5　printf 转换规范的组成部分

组件	功能描述
flags	总共有 5 个不同的 flag • # 使用替代格式输出。这取决于数据类型。对于 o（八进制数）类转换，输出结果以 0 开头。对于 x 和 X 类转换，输出则分别以 0x 和 0X 开头 • 0（零）用 0 填充输出。这代表着字段前会填充 0，如 000380 • -（破折号）输出左对齐。默认情况下，printf 是右对齐输出的 • （空格）为正数产生一个前导空格 • +（加号）正数符号。默认情况下，printf 只会输出负数的符号
width	一个数字，该数字指定了最小字段宽度
.precision	对于浮点数，便是指定小数点后的小数精确度。对于字符串转换，precision 则指定了输出字符的个数

表 21-6 列出了一些不同格式化实例。

表 21-6　print 转换规范实例

参数	格式	转换结果	说明
380	"%d"	380	对整数进行简单的格式化
380	"%#x"	0x17c	使用替代格式标记将整数格式化为一个十六进制数
380	"%05d"	00380	将整数格式化为至少 5 个字符宽度的字段，不足位数可在前面填充 0
380	"%05.5f"	380.00000	将数字格式化为精确到小数点后 5 位的浮点数，不足位数用 0 填充。由于指定的最小字段宽度（5）要比格式化后实际的数值位数小，所以此处并没有进行填充
380	"%010.5f"	0380.00000	把最小字段宽度增加为 10，并且 0 填充可见
380	"%+d"	+380	+标记符表示此数是正数
380	"%-d"	380	-标记表示左对齐
abcedfghijk	"%5s"	abcedfghijk	用最小的字段宽度来最小化字符串
abcedfghijk	"%.5s"	abcde	根据字符串的精确位数截断字符串

同样，printf 通常用于脚本文件的表格数据格式化操作，而并不会直接应用于命令行。不过，我们仍然可以用其解决各种各样的格式化问题。首先，我们可以利用 printf 输出一些由制表符隔开的字段。

```
[me@linuxbox ~]$ printf "%s\t%s\t%s\n" str1 str2 str3
str1    str2     str3
```

插入\t（Tab 的转义符），便获得了理想结果。接下来，利用其输出一些格式整齐的数字。

```
[me@linuxbox ~]$ printf "Line: %05d %15.3f Result: %+15d\n" 1071 3.14156295
32589
Line: 01071           3.142 Result:          +32589
```

此例充分显示了最小字段宽度对字段间距的影响。那么如何格式化一个小网页呢？

```
[me@linuxbox ~]$ printf "<html>\n\t<head>\n\t\t<title>%s</title>\n\t</head>
\n\t<body>\n\t\t<p>%s</p>\n\t</body>\n</html>\n" "Page Title" "Page Content"
<html>
        <head>
                <title>Page Title</title>
        </head>
        <body>
                <p>Page Content</p>
        </body>
</html>
```

21.2 文档格式化系统

到目前为止，本章只介绍了一些简单的文本格式化工具。这些工具对于一些小的、简单的任务非常有用，那对于一些比较大的任务呢？系统为用户提供了多种编辑多类型文件的工具，特别是科学类和学术类的出版物，这也是 UNIX 成为一个在科学和技术领域比较受欢迎的操作系统的原因之一（还有一个原因就是 UNIX 提供了强大的多任务、多用户的软件开发环境）。事实上，正如下面的 GNU 文档所描述的，文档编辑对 UNIX 的开发很重要。

> 第一个版本的 UNIX 系统是在贝尔实验室的 PDP-7 上开发的。1971 年，开发者们想要一台 PDP-11 以对操作系统进行进一步的研究。为了充分证明这一系统值得花费这么多资金，开发者们指出会为 AT&T 专利部门完成一种文件格式化系统。J.F. Ossanna 是该格式化程序首任作者，另外，此程序其实是 McIllroy 编写的 *roff* 程序的重新实现。

21.2.1 roff 和 T_EX 家族

当今主宰文档格式化领域的主要有两大家族，它们是从原始 roff 程序延伸而来的 nroff 和 troff 和基于 Donald Knuth 的 T_EX（发音为‘tek’）排版系统。自然，T_EX 中间偏低的字母“E”不可忽略。

roff 这个名字来源于“I'll run off a copy for you.”中的短语“*run off*”。nroff 程序格式化后的文档一般输出至使用等宽字体的设备，诸如字符终端、类打字机风格的打印机等。在该程序风靡期间，它几乎覆盖了所有与计算机相连的打印设备。后来的 troff 程序则用于格式化排字机等设备输出的文档，这些设备通常输出“camera-ready”格式的文本用于商业印刷。如今，多数电脑打印机都能模拟排字机的输出。roff 系列还包含了一些用于处理其他文件类型的程序，这些程序包括 eqn（针对数学等式）以及 tbl（针对表格）等。

T_EX 系统（其稳定版本）在 1989 年首次成形，并且一定程度上，取代了 troff 而成为排字机输出文档的格式化工具。本书不讨论 T_EX，不仅是因为其比较复杂（有好几本单独讲述 T_EX 的书籍），还因为多数现代 Linux 系统并不会默认安装它。

注意　对于那些对安装 TEX 有兴趣的读者，可以学习 texlive 软件包以及 LyX 图形编辑器的相关知识，texlive 软件包在绝大多数发行版本库中都能找到。

21.2.2 groff——文档格式化系统

groff 其实是 GNU 实现方式的 troff 系列程序集，它还包含一个用于模拟 nroff 及其他 roff 家族系列的程序功能的脚本。

虽然 roff 及其衍生体都是用来创建格式化文件的，但是它们格式化的方式对于现代用户来说却非常陌生。如今多数文档都是用文字处理器编辑的，并且这些文字处理器可以一步完成文档的内容编辑和布局格式安排。图形化文字处理器出现之前，文档编辑通常包括两个步骤——使用文本编辑器编辑内容和使用诸如 troff 这样的程序进行格式化，格式化程序的指令都已经通过标记语言嵌入到文本中。现代的网页制作过程与此过程相似，因为网页也是先通过某种文本编辑器编辑，然后使用由网络浏览器提供的 HTML 标记语言来描述最后的网页布局的。

当然，本章并不打算全面讨论 groff，因为它的许多标记语言元素都与一些非常诡秘的排版细节相关。相反，大家还是应该把精力集中到仍然广泛应用的“宏包”上。这些“宏包”将许多低级命令压缩成一个很小的高级命令集，从而

使得 groff 使用起来容易得多。

耽误一点时间，大家先来看一看下面这个不起眼的手册页，它是*/usr/share/man* 目录下的一个 gzip 压缩的文本文件。查看其解压缩后内容，可以用下面的命令行（ls 的 man 手册页在第一部分有说明）。

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | head
.\" DO NOT MODIFY THIS FILE! It was generated by help2man 1.35.
.TH LS "1" "April 2008" "GNU coreutils 6.10" "User Commands"
.SH NAME
ls \- list directory contents
.SH SYNOPSIS
.B ls
[\fIOPTION\fR]... [\fIFILE\fR]...
.SH DESCRIPTION
.\" Add any additional description here
.PP
```

与其正常的 man 手册页进行比较分析，就可以发现标记语言及其输出结果之间的相关性。

```
[me@linuxbox ~]$ man ls | head
LS(1)                    User Commands                    LS(1)


NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...
```

这真的很有趣，因为此 man 手册页是由 groff 使用 mandoc 的“宏包”进行浏览的。事实上，我们还可以用如下管道传输模拟 man 命令。

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | groff -mandoc -T ascii |
head
LS(1)                    User Commands                    LS(1)



NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...
```

此处，我们利用了 groff 程序，并设置参数指定为 mandoc“宏包”以及 ASCII 的输出格式。groff 可以输出多种不同格式的结果，如果未指定输出格式，那么 PostScript 便是其默认输出格式。

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | groff -mandoc | head
%!PS-Adobe-3.0
```

```
%%Creator: groff version 1.18.1
%%CreationDate: Thu Feb  2 13:44:37 2012
%%DocumentNeededResources: font Times-Roman

%%+ font Times-Bold
%%+ font Times-Italic
%%DocumentSuppliedResources: procset grops 1.18 1
%%Pages: 4
%%PageOrder: Ascend
%%Orientation: Portrait
```

PostScript 是一种页面描述语言，一般用于描述送至类排字机设备打印的页面内容。我们可以提取 groff 命令的输出结果并将其存于文件中（假设使用的是备有 Desktop 目录的图形化桌面系统）。

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | groff -mandoc > ~/Desktop
/foo.ps
```

桌面上应该会出现一个表示输出文件的图标，双击该图标启动页面浏览器，浏览器便以其所默认的形式（见图 21-1）显示该文件内容。

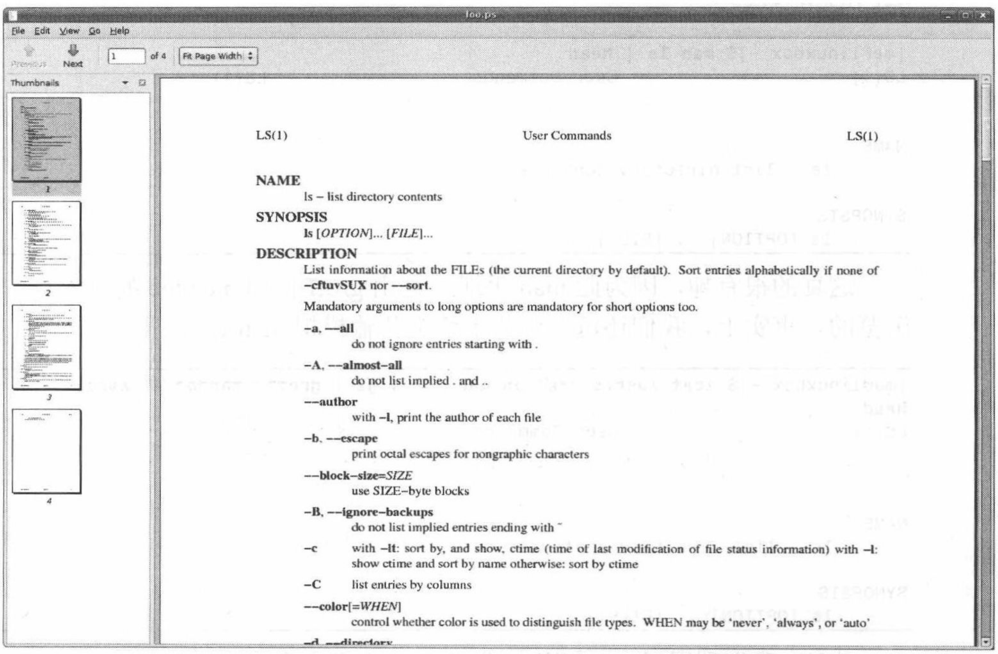


图 21-1　用 GNOME 中的页面浏览器查看 PostScript 格式的输出内容

我们得到的是一个排版整齐的 ls 的 man 手册页！事实上，我们可以用下面的 ps2pdf 命令将 PostScript 格式的文件转化为 PDF（*Portable Document Format*，可移植文档格式）文件，示例如下。

```
[me@linuxbox ~]$ ps2pdf ~/Desktop/foo.ps ~/Desktop/ls.pdf
```

ps2pdf 程序是 ghostscript 软件包的一个小成员，该程序在多数支持打印的 Linux 系统上都有安装。

注意 Linux 系统通常包含许多进行文件格式转换的命令行程序，它们的命名方式通常为 format2format。尝试使用命令行 ls /usr/bin/*[[:alpha:]]2[[:alpha:]]*输出程序名列表，另外作为比较，可以尝试搜索名为 formattoformat 的程序。

作为 groff 的最后一个演示实例，重新启用“老朋友” *distros.txt* 文件。此次，使用表格格式化工具 tbl 对 *distros.txt* 文件中的 Linux 发行版本列表进行排版。要完成这样的操作，需要使用较早的 sed 脚本向 groff 将要处理的文本流中增加标记语言。

首先，需要对 sed 脚本做一些修改，比如增加 tbl 命令需要的一些必要项。使用文本编辑器，将 *distros.sed* 脚本改为如下内容。

```
# sed script to produce Linux distributions report

1 i\
.TS\
center box;\
cb s s\
cb cb cb\
l n c.\
Linux Distributions Report\
=\
Name Version Released\
_
s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
$ a\
.TE
```

请注意，该脚本文件若要正常工作，必须确保其中的 *“Name Version Released”* 词组之间是以 tab 制表符而不是空格分开。此外，将输出结果另存为 *distros-tbl.sed* 文件。tbl 使用.TS 和.TE 请求作为表格的开头和末尾，跟在.TS 请求后面的行定义了表格的全局属性。就本例而言，其定义了页面内容水平居中并用文本框包围这样的属性。定义文本的其余内容则描述了每个表格中行的布局。现在，我们使用新的 sed 脚本处理 groff 的格式化结果，便会得到下面的内容。

```
[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-tbl.sed | groff
-t -T ascii 2>/dev/null
              +------------------------------+
              | Linux Distributions Report   |
              +------------------------------+
              | Name     Version    Released |
              +------------------------------+
              |Fedora      5       2006-03-20 |
```

```
|Fedora      6        2006-10-24    |
|Fedora      7        2007-05-31    |
|Fedora      8        2007-11-08    |
|Fedora      9        2008-05-13    |
|Fedora     10        2008-11-25    |
|SUSE       10.1      2006-05-11    |
|SUSE       10.2      2006-12-07    |
|SUSE       10.3      2007-10-04    |
|SUSE       11.0      2008-06-19    |
|Ubuntu      6.06     2006-06-01    |
|Ubuntu      6.10     2006-10-26    |
|Ubuntu      7.04     2007-04-19    |
|Ubuntu      7.10     2007-10-18    |
|Ubuntu      8.04     2008-04-24    |
|Ubuntu      8.10     2008-10-30    |
+-----------------------------+
```

groff 增加-t 选项表示先用 tbl 预处理文本流。同样，增加-T 选项则输出 ASCII 格式的数据，否则会输出默认的 PostScript 格式。

如果显示终端或是打字机类型的打印机能力有限，那么 ASCII 码的输出格式其实是所能期望的最好的显示格式了。但是，如果指定了 PostScript 格式的输出，并用图形化方式查看，便会得到令人更满意的输出效果（见图 21-2）。

```
[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-tbl.sed | groff
-t > ~/Desktop/foo.ps
```

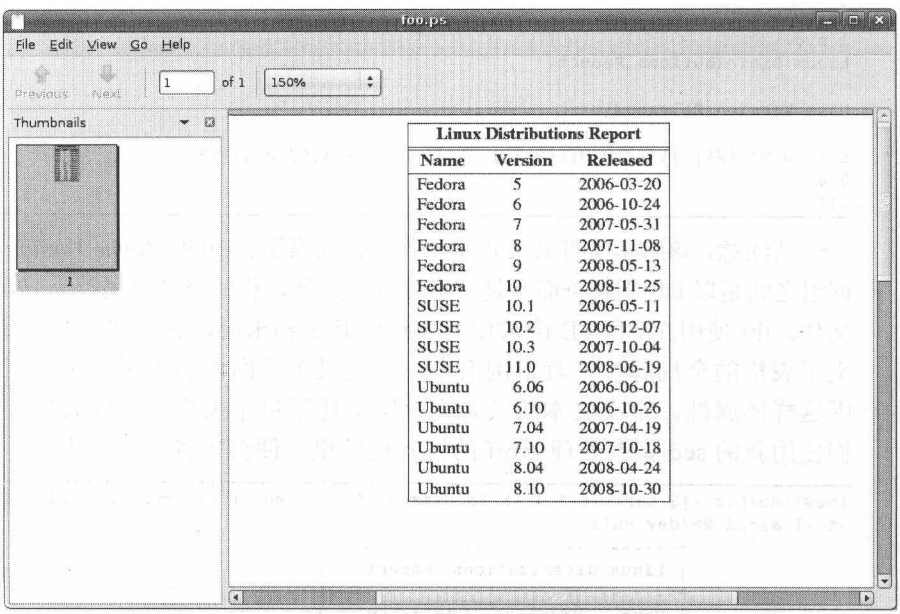


图 21-2　完成格式转换后的输出表格

21.3 本章结尾语

既然文本对于类 UNIX 操作系统中的字符是如此重要，那么有许多用于操控和格式化文本的工具就显得很有必要。我们已经知道，确实有很多这样的文本工具！简单的格式化工具，如 fmt 和 pr 在一些生成短文档的脚本中用途很大，而像 groff 这类复杂工具则可以用来排版书籍。当然，我们可能永远也不会用命令行工具来编辑技术文档（尽管也有许多人这么做！），但是知道可以用命令行完成也不失为一件好事。

第22章 打印

前面几章讲述了文本的操纵，现在是时候讨论如何将文本输出到纸张上了，本章将会介绍一些用于打印文件以及控制打印机操作的命令行工具。诚然，本书并不会讨论如何配置打印参数，因为这取决于不同的发行版本，而且通常操作系统安装时就已自动设置完成。请留意，本章的实例练习中会需要一个有效的打印机配置文件。

本章会讨论如下命令。

- pr：转换文本文件，从而进行打印操作。
- lpr：打印文件。
- lp：打印文件（System V）。
- a2ps：格式化文件，以在 PostScript 打印机上打印。
- lpstat：显示打印状态信息。
- lpq：显示打印机队列状态。

- lprm：取消打印任务。
- cancel：取消打印任务（System V）。

22.1 打印操作简史

在充分理解类 UNIX 操作系统的打印特性之前，我们必须首先了解一些打印操作的发展史。类 UNIX 系统的打印操作可以追溯到操作系统的起源。那个时期，打印机及其使用方法都与现在有很大区别。

22.1.1 灰暗时期的打印

与计算机类似，打印机在前 PC 时代也具有庞大、昂贵和集中化的特点。20 世纪 80 年代，计算机用户都在离计算机一定距离的终端上进行操作，打印机也是置于计算机附近而处于计算机操作员的监控下工作。

UNIX 早期，打印机贵而集中化，多个用户共用一台打印机是司空见惯的事。为了区分某个任务属于哪一个具体用户，一张显示用户名的标题页（banner page）通常会在每个打印任务的开头打印出来，然后计算机辅助人员会把当天的打印任务装到一个手推车中，然后将它们配送给各个用户。

22.1.2 基于字符的打印机

与今天的打印机技术相比，20 世纪 80 年代的打印机技术在两个方面有很大的不同。第一，20 世纪 80 年代的打印机基本上都是击打式打印机。击打式打印机采取机械机制通过色带撞击页面，从而在页面上形成字符印迹。菊花轮打印和点阵打印是当时非常受欢迎的两种技术。

第二，也是更重要的一点，早期打印机使用的是设备本身所固有的字符集。例如，菊花轮打印机只能打印那些已经塑造成菊花轮花瓣形状的字符，这使得打印机就像高速打字机一样。而多数打字机打字的时候使用的是等宽（固定宽度）字体，这意味着每个字符都有相同的宽度。并且打印机在纸张的固定位置进行打印，打印区域包含固定数量的字符。多数打印机水平方向上每英寸打印 10 个字符（10CPI），垂直方向上每英寸打印 6 行（6LPI）内容。在这样的机制下，一张美式信纸可容纳 85 字符宽、66 行高的内容。考虑到每个边界将留有少许页边距，于是每行最多可打印 80 个字符，这就解释了为什么终端显示（以及终端仿真器）只有 80 个字符宽。它提供了一种以等宽字体表现的 WYSIWYG（所见即所得）的输出视觉效果。

需要打印的字符数据以简单的字节流的形式发送给上述类打字机打印设备。例如，在打印字母 a 时，ASCII 码 97 便被发送出去。另外，低编号的 ASCII 控制码则用于移动打印机的送纸箱和纸张，以及取代回车、换行、换页符等。控制代码的使用，可以获得一定的字体效果。比如粗体可以通过先打印一个字符，然后退格再打印一次的方法来实现。如下例，我们使用 nroff 浏览 man 手册页，并且使用 cat –A 选项查看粗体效果。

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | nroff -man | cat -A | head
LS(1)                   User Commands                   LS(1)
$
$
$
N^HNA^HAM^HME^HE$
       ls - list directory contents$
$
S^HSY^HYN^HNO^HOP^HPS^HSI^HIS^HS$
       l^Hls^Hs[_^HO_^HP_^HT_^HI_^HO_^HN]...[_^HF_^HI_^HL_^HE]...$
```

其中^H（表示 Ctrl-H 快捷键）字符是用来创建粗体效果的后退格。类似地，我们还可以用退格/下划线以产生下划线效果。

22.1.3 图形化打印机

GUI 的发展促进了打印机技术的重大变革。由于计算机日渐趋向于图形化显示，打印技术也从基于字符走向图形化。低成本的激光打印机的出现为这一转变提供了可能。激光打印机可以在页面的任何可打印区域打印一个个小点，而不是打印单个固定字符，如此便可以按比例打印文字（类似于排版机），甚至是照片以及高质量的图表。

然而，基于字符的打印机制向图形化机制的转变仍存在一个艰巨的技术性挑战。原因是，使用基于字符的打印机时，填充一页纸所需要字节数可以用如下方法计算（假定一页有 60 行，每一行包含 80 个字符）：60 × 80 = 4,800 字节。

相比较而言，一个每英寸 300 个点（300 DPI）的激光打印机（假设每页包含 8 × 10 英寸打印区域）则需要(8 × 300) × (10 × 300) ÷ 8 = 900,000 个字节来填充一页纸。

多数慢速 PC 网络根本无法处理激光打印机打印一整页纸所需要的这近 1MB 的数据，所以很明显，我们需要一个更高端的发明。

页面描述语言（PDL）解决了这一问题，页面描述语言是一种描述页面内容的编程语言。它的描述方式可以简单表达为“在这个地方，使用 10 个点的无衬线字体（Helvetica,），打印 a 字符；在这个地方……”，直到该页所有内容被

描述结束。第一个主流页面描述语言是 Adobe 系统的 PostScript，如今仍被广泛使用。PostScript 几乎是为排版或是其他的图表图像印刷而量身定做的一套完整的编程语言。它除了自带的 35 种标准的高质量字体外，还允许用户在运行时定义额外字体。对 PostScript 的支持已内嵌在打印机中，这解决了数据传送的问题。虽然 PostScript 语言相比于基于字符的打印机使用的简单比特流显得冗长，但却比打印整页所需要的字符容量要小很多。

PostScript 打印机以 PostScript 程序作为输入。该打印机自带处理器和存储器（这些使得打印机成为比其连接的计算机功能更强的计算机），它可以运行 PostScript 解释器，该解释器读入 PostScript 程序并将其解释结果送至打印机内部的存储器，如此便形成了向纸张传送的位（点）模式。这个将程序转变为大型位模式（称为点阵图）的过程统称为光栅图形处理器，或 RIP。

随着时间的推移，计算机和网络的速度也越来越快，这使得可以让计算机来执行 RIP 过程而非让打印机执行，这使得高质量打印机的价格下降了很多。

如今的许多打印机仍然支持字符流输入，但是那些价钱比较低的打印机仍然不可以。它们主要依赖于主计算机的 RIP 提供比特流以进行点打印。当然，现今也还是有一些 PostScript 打印机的。

22.2 Linux 方式的打印

现代 Linux 系统采取两组软件来执行和管理打印操作：第一种是 CUPS（通用 UNIX 打印系统），提供打印驱动程序以及打印任务管理程序；另一种是 Ghostscript，这是一个 Postscript 编译器，充当 RIP 的作用。

CUPS 通过创建、维护打印队列进行打印机管理。正如前面在介绍打印机历史时介绍的，UNIX 打印的设计初衷是管理多用户共享的集中化打印机。由于打印机的处理速度比驱动它们的计算机慢，所以打印系统需要一种协调多个打印任务的机制，从而使所有事情能井然有序地进行。CUPS 能够识别不同类型的数据（在合理范围内），且能将文件转换为可打印的形式。

22.3 准备打印文件

作为命令行用户，我们多数还是倾向于打印文本文件，虽然其他的数据形式也可以打印。

22.3.1 pr——将文本文件转换为打印文件

前面章节略微介绍了 pr 的相关知识，本节将讨论其与打印操作联合使用时用到的一些参数选项。在前面打印机简史这一节中，我们介绍了基于字符的打印机使用等宽字体使得每页有固定数目的行并且每行有固定的字符数。pr 则用于调整文本以适应特定纸张的大小，并且可自主选择页眉和页边距。下表 22-1 总结了较常使用的一些选项。

表 22-1 常见的 pr 选项

选项	功能描述
+first[:last]	输出一个从 first 开始以 last 结束的页范围
-columns	将页的内容分成指定的 columns 列
-a	默认情况下，多列输出是垂直列出的。通过增加-a（across）选项，内容便是水平列出
-d	隔行打印输出
-D format	用 format 格式来格式化页眉的显示日期。可以查看日期命令的 man 手册页以了解格式式字符串的描述
-f	使用换页符而不是回车符作为页与页之间的分隔符
-h header	在页眉的中间部分，使用 header 代替正在处理的文件名
-l length	将页的长度设为 length。默认是 66 行（按每英寸有 6 行的美式字母算）
-n	对行进行编号
-o offset	创建一个有 offset 字符宽的左页边距
-w width	设定页面宽度为 width，默认是 72 个字符

pr 通常用于管道传输的过滤器。下面例子生成了一个/usr/bin 文件夹下的目录列表，并用 pr 将其格式化为分页的、三列的输出。

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 -w 65 | head

2012-02-18 14:00                                    Page 1
[                        apturl                   bsd-write
411toppm                 ar                       bsh
a2p                      arecord                  btcflash
a2ps                     arecordmidi              bug-buddy
a2ps-lpr-wrapper         ark                      buildhash
```

22.4 向打印机发送打印任务

CUPS 打印组件支持两种打印方法，它们都是类 UNIX 系统过去使用过的。

一种方法叫做 Berkeley 或 LPD（UNIX 的 Berkeley 软件发行版本中使用的），运用的是 lpr 命令；另外一种方法叫做 SysV（来源于 UNIX 的 System V 版本），运用的是 lp 命令。这两个命令大致做着相同的事情，用户可依据个人喜好选择使用。

22.4.1 lpr——打印文件（Berkeley 类型）

lpr 命令可以将文件传送至打印机，同时由于其支持标准输入，所以也可用于管道传输。例如，要打印上述的多列目录列表，我们可以进行如下操作。

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 | lpr
```

该报告将被送至系统默认的打印机。如果想将文件发送至不同的打印机，可以使用-P 选项，示例如下。

```
lpr -P printer_name
```

此处 printer_name 是指目标打印机的名称。可以用下面的命令行查看系统打印机列表。

```
[me@linuxbox ~]$ lpstat -a
```

注意 许多 Linux 发行版本可以定义一个“虚拟打印机”，以输出 PDF 格式的文件，而不是在真正的物理打印机上打印，这一特性可以方便地用于实践打印命令。检查打印机设置程序可以判断系统是否支持这一设置。对于某些发行版本，你可能会需要安装额外的软件包（比如像 cups-pdf 软件包）以支持这一特性。

表 22-2 列出了 lpr 的常用选项。

表 22-2 常用的 lpr 选项

选项	功能描述
-# number	打印 number 份副本
-p	每一页都将包括日期、时间、工作名称和页码的页眉用阴影打印出来。这种所谓的“优质打印”选项可以用于打印文本文件
-P printer	指定用于打印输出的打印机名。如果未指定打印机，那就是用系统默认的打印机
-r	打印结束后删除文件。此选项适用于那些产生临时打印输出文件的程序

22.4.2 lp——打印文件（System V 类型）

与 lpr 类似，lp 命令既支持文件输入也支持标准输入。它与 lpr 的不同之处在于它有一个不同（稍微复杂点）的参数选项设置。表 22-3 列出了常用的选项。

表 22-3 常见的 lp 选项

选项	说明
-d printer	设置目标打印机为 printer。如果未指定-d 选项，将会使用系统默认的打印机
-n number	打印 number 份副本
-o landscape	将输出设置为横向
-o fitplot	根据页面大小缩放文件。这在打印诸如 JPEG 文件时非常有用
-o scaling=number	设定文件缩放比例为 number。如果该值为 100，则正好填充一页纸；如果该值小于 100，那么一页纸将填不满；如果该值大于 100，则打印内容将打印在多个页面上
-o cpi=number	设置每英寸输出字符数位 number。默认是 10
-o lpi=number	设定每英寸输出指定 number 的行，默认是 6
-o page-bottom=points -o page-left=points -o page-right=points -o page-top=points	设置页边距。其值以点的形式表示，点是排版测量单位，每英寸有 72 个点
-P pages	指定页列表。页的表达形式可以为用逗号隔开的页列表或是以"—"表示的页范围，如 1,3,5,7-10

让我们重新生成目录清单，此次打印的规格为 12CPI（字符/英寸）和 8LPI（行/英寸），并且左边距为 0.5 英寸。请注意，考虑到需要适应新的页面大小，所以必须调整 pr 的参数选项。

```
[me@linuxbox ~]$ ls /usr/bin | pr -4 -w 90 -l 88 | lp -o page-left=36 -o cpi=
12 -o lpi=8
```

该管道使用了比默认规格更小的字体从而得到了一个 4 列列表。每行增加的字符数使得一页纸中能够容纳更多列。

22.4.3 另外一个参数选项：a2ps

a2ps 是一个非常有趣的命令，从其命令名中可以看出它是一个格式转换程序，但其功能远不止这些。该名称原义是指将 ASCII 格式转化为 PostScript 格式，并且用于为 PostScript 打印机上的打印任务准备文本文件。然而，随着时间的推

移，该程序的功能不断扩大，至今已可以将任何格式的文件转化为 PostScript 格式了。尽管表面上看起来它是一个格式转换程序，但其实际是一个打印程序。它会将默认输出而非标准输出送至系统默认的打印机。该程序默认使用的是“优质打印机”模式，也就是说会自动优化输出内容的可视性。我们可以用该程序在桌面上创建一个 PostScript 文件。

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 -t | a2ps -o ~/Desktop/ls.ps -L 66
[stdin (plain): 11 pages on 6 sheets]
[Total: 11 pages on 6 sheets] saved into the file '/home/me/Desktop/ls.ps'
```

该命令行用 pr 过滤了该文本流，使用-t 选项（省略了页眉页脚），然后使用 a2ps 指定了一个每页 66 行（-L 选项）的输出文件（-o 选项）以匹配 pr 的分页输出。如果我们选用合适的文件浏览器查看其输出结果，可以看到下图 22-1 的内容。

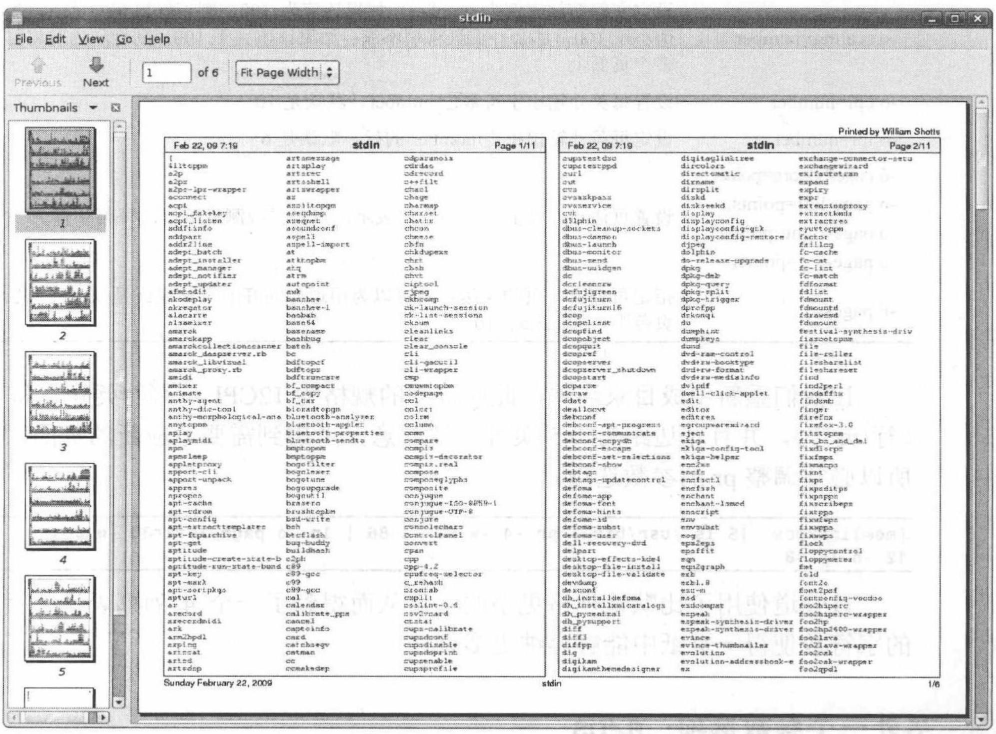


图 22-1　a2ps 命令的输出内容

可以看到，默认的输出布局是“双面”（“two up”）格式，也就是两页纸的内容会在一张纸的正反面打印。a2ps 会自动应用美观的页眉和页脚。

a2ps 有许多选项，表 22-4 总结了一些。

表 22-4 a2ps 选项

选项	功能描述
--center-title text	将中心页标题设为 text
--columns number	将页分成 n 列。默认值是 2
--footer text	设置页脚内容为 text
--guess	给出输入文件的类型。因为 a2ps 试图转换以及格式化所有类型的数据，该选项可以用于在给定了具体文件的情况下，告诉 a2ps 下一步做什么
--left-footer text	将左页脚内容设为 text
--left-title text	将左页标题设置为 text
--line-numbers=interval	每 interval 的间隔对输出文件进行编号
--list=defaults	显示默认设置
--list=topic	显示 topic 的设置，这里的 topic 可以是下面的一种：委托（用于转换数据的外部程序）、编码、属性、变量、媒介（纸张大小及其类似的设置）、ppd（Postscript 打印机描述）、打印机、序言（位于正常输出前面的代码段）、样式表、用户选项
--pages range	打印 range 范围内的内容
--right-footer text	设置右页脚内容为 text
--right-title text	设置右页标题内容为 text
--rows number	将页面分成 number 行。默认值是 1
-B	无页眉
-b text	设置页眉内容为 text
-f size	使用 size 点的字体
-l number	设置每行有 number 个字符。该选项和下面的-L 选项可以用于诸如 pr 等的其他命令的分页处理，以适应打印纸张的大小
-L number	设置每页有 number 行
-M name	使用名为 name 的纸张大小，例如 A4
-n number	打印 n 份副本
-o file	将输出内容送至 file。如果指定 file 是 1，则是标准输出
-P printer	使用 printer 打印机，如果并未指定任何打印机，那么就使用系统的默认打印机
-R	纵向排版
-r	横向排版
-T number	设置制表符 tab 作为每一个数字字符的结尾
-u text	采用底图（水印）技术打印文本纸张

该表仅仅是一个简单的概括，a2ps 还有很多其他选项。

注意 人们仍在积极开发 a2ps 程序。就我使用 a2ps 的经历而言，不同 Linux 发行版本中 a2ps 有着不同的表现。在 CentOS 4 系统上，默认输出为标准输出。在 CentOS 4 和 Fedora 10 系统上，尽管程序已经设置默认使用信封大小的纸张，但其默认输出仍为 A4 大小的纸张。诚然，这些问题可以通过详细地定义所需要的参数选项来解决。在 Ubuntu 8.04 系统上，a2ps 则表现得如前面所记录的一样。

还要注意的是，有另外一种将文本转化为 PostScript 的工具叫做 enscript。它可以进行许多与 a2ps 同样的格式化和打印操作，不同的是它只支持文本输入。

22.5 监测和控制打印任务

与 UNIX 打印系统类似，CUPS 也是以处理来自多用户的多项打印任务而设计的。每个打印机都有一个打印队列，打印任务排列其中等待被送至打印机打印。CUPS 提供了几个用于管理打印机状态和打印队列命令行的程序。与 lpr 和 lp 程序类似，这些管理程序都是以 Berkeley 和 System V 打印系统相应的程序为原型的。

22.5.1 lpstat——显示打印系统状态

lpstat 程序可以确定系统中打印机的名称和可用性。例如，假定一台打印系统，既有物理打印设备（叫做 printer），也有 PDF 虚拟打印机（叫作 PDF），那么便可以用下面的命令行检查它们各自的状态。

```
[me@linuxbox ~]$ lpstat -a
PDF accepting requests since Mon 05 Dec 2011 03:05:59 PM EST
printer accepting requests since Tue 21 Feb 2012 08:43:22 AM EST
```

更进一步，我们可以用下面的方式得到更具体的打印系统配置描述。

```
[me@linuxbox ~]$ lpstat -s
system default destination: printer
device for PDF: cups-pdf:/
device for printer: ipp://print-server:631/printers/printer
```

本例中，printer 是系统默认的打印机，同时它也是一个利用网络打印协议（ipp://）与一个叫做 print-server 的系统连接的网络打印机。

表 22-5 中描述了 lpstat 的常用选项。

表 22-5　常用的 lpstat 选项

选项	功能描述
-a [printer...]	显示 printer 打印机的打印队列状态。请注意，该状态显示的是打印队列接受打印任务的能力，而不是物理打印机的状态。如果未指定打印机，所有的打印队列都会显示出来
-d	显示系统默认的打印机名
-p [printer...]	显示指定的 printer 状态。如果未指定打印机，所有的打印机都会显示出来
-r	显示打印服务器的状态
-s	显示状态汇总报告
-t	显示一个完整的状态报告

22.5.2 lpq——显示打印队列状态

我们可以使用 lpq 程序查看一个打印队列的状态，该程序可以查看打印机队列状态及其所包含的打印任务。如下为系统默认的打印机 printer 的一个空队列。

```
[me@linuxbox ~]$ lpq
printer is ready
no entries
```

如果事先并未指定打印机（使用-P 选项），系统便会显示默认的打印机。如果向打印机发送打印任务，然后查看打印队列，便会看到如下列表。

```
[me@linuxbox ~]$ ls *.txt | pr -3 | lp
request id is printer-603 (1 file(s))
[me@linuxbox ~]$ lpq
printer is ready and printing
Rank    Owner  Job    File(s)                     Total Size
active  me     603    (stdin)                     1024 bytes
```

22.5.3 lprm 与 cancel——删除打印任务

CUPS 提供了两个用于终止打印任务并将它们从打印队列中移除的程序。其中一个就是 Berkeley 类型的 lprm，另外一个是 System V 的 cancel。它们在所支持的参数选项上有稍许不同，但基本上实现的是同样的功能。以上面例子中使用的打印任务为例，将该打印任务终止并移除，可以使用如下命令行。

```
[me@linuxbox ~]$ cancel 603
[me@linuxbox ~]$ lpq
printer is ready
no entries
```

这两个命令都提供了一些选项，用于移除属于一个特定用户、特定打印机以及多个任务号的所有打印任务，它们各自的 man 手册页都详细介绍了其用法。

第23章 编译程序

本章将介绍如何通过源代码生成可执行程序。开放源代码是 Linux 自由开源的必要因素，整个 Linux 系统的开发依赖于开发人员之间的自由交流。对于多数桌面系统用户来说，编译已经是一门失传的艺术。编译技术虽然曾经非常普遍，但是如今，版本发行商却维护着很大的预编译二进制库，以便用户下载使用。在本书编写之时，Debian 系统的预编译库（所有 Linux 发行版中最大的库之一）包含了近 23000 个软件包。

那么，为什么要编译软件呢？有如下两个原因。

- **可用性**：尽管有些发行版已经包含了版本库中的一些预编译程序，但并不会包含用户所有可能需要的应用程序。这种情况下，用户获取所需要软件的唯一方式就是编译源代码。
- **及时性**：虽然有些发行版本专注于一些前沿的程序版本，但是多数并不会。这就意味着想要获取最新版本的程序，编译是必不可少的。

编译软件源代码是一项非常复杂并而有技术性的任务，这远远超出了多数用户

的能力范围。然而，仍有许多非常简单的编译任务，只需要几个步骤即可完成。编译的复杂程度完全取决于所要安装的软件包。本章会介绍一个非常简单的例子，从而让大家对编译过程有一个大致了解，并为那些准备深入学习的人打下基础。

本章会介绍一个新的命令。

- make—维护程序的工具。

23.1 什么是编译

简单来说，编译就是一个将源代码（由程序员编写的人类可读的程序描述）翻译成计算机处理器能识别的语言的过程。

计算机处理器（或 CPU）在一个非常基础的层次上工作，只能运行称之为机器语言的程序。而机器语言其实就是一些数值代码，它描述的都是一些非常小的操作，比如“增加某个字节”、“指向内存中某个位置”或者“复制某个字节”等，并且每一个这样的指令都是以二进制（0 和 1）的形式表示的。最早的计算机程序就是用这样的数值代码编写，这也许可以解释为什么写这些数值代码的程序员吸很多烟，喝大量咖啡，戴着厚重的眼镜。

汇编语言的出现解决了这一问题，因为它用诸如 CPY（复制）和 MOV（转移）等更容易的助记符取代了那些数值代码，汇编语言编写的程序会由汇编程序（assembler）处理成机器语言。如今，汇编语言仍然用于某些专门的编程任务，诸如设备驱动和嵌入式系统等。

之后出现了高级编程语言，被称为高级语言是因为它们可以让程序员少关注些处理器的操作细节而把更多的精力集中在解决手头的问题上。早期（20 世纪 50 年代开发的）的高级语言包括 FORTRAN（专为科学技术任务设计）和 COBOL（专为商务应用设计），两者至今仍然在某些专门领域发挥着作用。

虽然现在流行的编程语言有很多，但有两个占据着主导地位，C 和 C++便是多数现代系统采用的编程语言。下面的内容中，我们编译了一个 C 语言程序作为例子进行讲解。

高级语言编写的程序通过编译器转换为机器语言。有些编译器则将高级语言程序转换为汇编语言，然后再使用一个汇编程序（assembler）将其转换为机器语言。

经常与编译一起使用的步骤就是链接。程序执行着许多共同的任务。例如，打

开一个文件，许多程序都会需要进行此操作。如果每个程序都采用自己的方式实现该功能的话便是一种浪费。编写一个用于打开文件的单个程序，并允许其他程序共享它，反而更有意义。提供这种通用任务支持功能的便是库，库中包含了多个例程，每一个实现的都是许多程序能够共享的通用任务。在/lib 和/usr/lib 目录中，我们可以发现很多这样的程序。链接器（linker）程序可以实现编译器的输出与编译程序所需要库之间的链接。该操作的最后结果就是生成一个供使用的可执行文件。

23.2 是不是所有的程序都需要编译

答案是否定的。我们都知道，有些程序，如 shell 脚本，可以直接运行而不需要编译，这些文件都是用脚本或解释型语言编写的。这些语言在近些年来越来越受欢迎，其中就有 Perl、Python、PHP、Ruby 以及其他多种语言等。

脚本语言由一个称为解释器的特殊程序来执行，解释器负责输入程序文件并执行其所包含的所有指令。通常来讲，解释型程序要比编译后的程序执行起来慢。这是因为在解释型程序中，每条源代码指令在执行时都要重新翻译一次该源代码指令。然而在编译后的程序中，每条源代码指令只翻译一次，并且该翻译结果将永久地记录到最后的可执行文件中。

那为什么解释型语言如此受欢迎？其实对于许多日常的编程工作，解释型程序的执行速度也是足够的，而其真正的优点在于开发解释型程序要比开发编译程序简单而迅速得多。程序开发总是经历着这样一个重复的循环——编码、编译和测试。随着程序规模的逐渐扩大，编译时间也逐渐变长。解释型程序则省略了编译这一过程，因此加速了程序的开发。

23.3 编译一个 C 程序

现在我们可以编译程序了。然而，在执行编译操作前，需要一些工具，诸如编译器、链接器以及 make 等。gcc（GNU C 编译器）是 Linux 环境中通用的 C 编译器，最初是由 Richard Stallman 编写的。多数 Linux 发行版本并不会默认安装 gcc，我们可以用下面的命令行查看系统是否安装了该编译器。

```
[me@linuxbox ~]$ which gcc
/usr/bin/gcc
```

该例子的结果表明已安装了此编译器。

注意　你所使用的 Linux 版本也许提供了一个用于软件开发的元数据包（一个软件包集）。如果这样，你可以直接安装该软件包便可在系统上编译程序。但如果你的系统并未提供该元数据包，则只能自己安装 gcc 和 make 软件包。当然，针对多数发行版本，安装这两个软件包已经足够执行接下来的例程。

23.3.1 获取源代码

从一个叫做 diction 的 GNU 项目中选择一个程序进行编译练习，这个方便短小的程序一般用于检查文本文件的质量和写作风格，它非常小并且很容易生成。

依据惯例，我们首先创建一个 src 目录用于存放源代码，然后使用 ftp 下载源代码至该目录。

```
[me@linuxbox ~]$ mkdir src
[me@linuxbox ~]$ cd src
[me@linuxbox src]$ ftp ftp.gnu.org
Connected to ftp.gnu.org.
220 GNU FTP server ready.
Name (ftp.gnu.org:me): anonymous
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd gnu/diction
250 Directory successfully changed.
ftp> ls
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
-rw-r--r--    1 1003  65534   68940 Aug 28  1998 diction-0.7.tar.gz
-rw-r--r--    1 1003  65534   90957 Mar 04  2002 diction-1.02.tar.gz
-rw-r--r--    1 1003  65534  141062 Sep 17  2007 diction-1.11.tar.gz
226 Directory send OK.
ftp> get diction-1.11.tar.gz
local: diction-1.11.tar.gz remote: diction-1.11.tar.gz
200 PORT command successful. Consider using PASV.
150 Opening BINARY mode data connection for diction-1.11.tar.gz (141062
bytes).
226 File send OK.
141062 bytes received in 0.16 secs (847.4 kB/s)
ftp> bye
221 Goodbye.
[me@linuxbox src]$ ls
diction-1.11.tar.gz
```

注意　由于编译源代码时我们便是源代码的维护者，所以我们将其存放于~/src 目录中。发行版本自行安装的源代码一般安装于/usr/src 目录下，而面向多用户使用的源代码则通常安装在/usr/local/src 目录中。

可以我们看到，源代码通常以一个压缩的 tar 文件的形式存在，有时被称为 *tarball*，该文件包含了源代码树，即构成该源代码的目录及文件的组织框架。连接到 FTP 站点后，我们便可以查看可用的 tar 文件列表并挑选其中最新的版本进行下载。在 ftp 中使用 get 命令，即可将文件从 FTP 服务器复制到本地主机。

下载了该 tar 文件后，就必须对其进行解压缩，这是通过 tar 程序来完成的：

```
[me@linuxbox src]$ tar xzf diction-1.11.tar.gz
[me@linuxbox src]$ ls
diction-1.11       diction-1.11.tar.gz
```

注意 该 diction 程序与所有其他 GNU 项目的软件一样，都遵循一定的源代码打包标准，Linux 系统中许多其他可用的源代码同样遵循这样的标准。该标准规定，当源代码的 tar 文件解压缩后，会创建一个包含源代码树的目录，并且该目录以 project-x.xx 的格式命名，此格式包含了该项目的名称及其版本号。这一机制有助于实现不同版本之间同一程序的安装。然而，在解压之前先检查源代码树的布局也是很有必要的，因为有些项目并不会创建目录而是将所有文件直接送至当前目录，这可能会给原先井然有序的 src 目录造成混乱。为了避免这样的事情发生，我们可以使用下面的命令行检查 tar 文件的内容。

```
tar tzvf tarfile | head
```

23.3.2 检查源代码树

解压 tar 文件会产生一个新的目录——diction-1.11。该目录包含了源文件树，具体内容如下。

```
[me@linuxbox src]$ cd diction-1.11
[me@linuxbox diction-1.11]$ ls
config.guess  diction.c         getopt.c      nl
config.h.in   diction.pot       getopt.h      nl.po
config.sub    diction.spec      getopt_int.h  README
configure     diction.spec.in   INSTALL       sentence.c
configure.in  diction.texi.in   install-sh    sentence.h
COPYING       en                Makefile.in   style.1.in
de            en_GB             misc.c        style.c
de.po         en_GB.po          misc.h        test
diction.1.in  getopt1.c         NEWS
```

此解压缩包中包含了许多文件。与其他许多程序相似，GNU 项目的程序也都有提供如 README、INSTALL、NEWS 和 COPYING 等这些文档文件。这些文件包含的是程序的描述、安装步骤说明及其许可条款。在着手生成程序前，

认真阅读 README 和 INSTALL 文档是很有必要的。

此目录下的其他有趣文件便是这些以.c 和.h 结尾的文件。

```
[me@linuxbox diction-1.11]$ ls *.c
diction.c  getopt1.c  getopt.c  misc.c  sentence.c  style.c
[me@linuxbox diction-1.11]$ ls *.h
getopt.h  getopt_int.h  misc.h  sentence.h
```

软件包提供的两个 C 程序（style 和 diction）便是以.c 结尾，并且它们都被分成了多个模块。如今，这种将一些大的程序分成较小的、易于管理的小程序片的做法已是司空见惯。这些源代码文件都是普通的文本文件，可以用 less 查看。

```
[me@linuxbox diction-1.11]$ less diction.c
```

.h 文件是大家熟知的头文件。这些文件，同样也是普通的文本文件。头文件中包含了对源代码文件或库中的例程的描述。编译器在链接这些例程模块时，必须给它提供一个其所用到的所有模块的描述。因此，在 diction.c 的开头，可以看到如下文本行。

```
#include "getopt.h"
```

该文本行会指示编译器在读取 diction.c 中的源代码内容时先读取文件 getopt.h 中的内容，进而读取 getopt.c 中的内容。getopt.c 文件包含的是由 style 和 diction 程序所共享的例程模块。

在 getopt.h 的 include 语句上面，还可以看到一些其他 include 语句，如下所示。

```
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

这些都是用来引用头文件，但是它们引用的是那些不在当前源目录下的头文件。它们由系统提供，为每个程序的编译提供支持。如果查看/usr/include 目录，便会看到如下内容。

```
[me@linuxbox diction-1.11]$ ls /usr/include
```

该目录下的头文件在安装编译器时便已安装。

23.3.3 生成程序

大多数程序都是使用一个简单的两行命令来生成的。

```
./configure
make
```

configure 程序其实是源代码树下的一个 shell 脚本，它的任务就是分析生成环境。多数源代码都设计成可移植的。也就是说，源代码可以在多种类型的 UNIX 系统上生成，只是源代码在生成时可能需要经过细微的调整以适应各系统之间的不同。configure 同样会检查系统是否已经安装了必要的外部工具和组件。

接下来运行 configure。由于 configure 并不是存放于 shell 通常期望程序所在的目录下，所以必须显式告知 shell 有关 configure 的位置，我们可在命令前添加 "./" 目录符来实现这一目的。该符号表示 configure 程序位于在当前的工作目录下。

```
[me@linuxbox diction-1.11]$ ./configure
```

configure 会在检查以及配置 build 程序的同时输出许多相关信息。运行结束后，会输出如下类似内容。

```
checking libintl.h presence... yes
checking for libintl.h... yes
checking for library containing gettext... none required
configure: creating ./config.status
config.status: creating Makefile
config.status: creating diction.1
config.status: creating diction.texi
config.status: creating diction.spec
config.status: creating style.1
config.status: creating test/rundiction
config.status: creating config.h
[me@linuxbox diction-1.11]$
```

需要重点注意的是，这里并没有错误信息。如果有的话，该 configure 操作将以失败告终，并且不会生成可执行程序，直到所有的错误都被纠正过来。

可以看到，configure 在源目录中创建了几个新文件，其中最重要的就是 Makefile。Makefile 是指导 make 命令如何生成可执行程序的配置文件，如果没有该文件，make 便无法运行。Makefile 也是一个普通的文本文件，所以我们可以用 less 查看其内容。

```
[me@linuxbox diction-1.11]$ less Makefile
```

make 程序的作用其实就是输入 makefile（通常叫做 Makefile），该文件描述了生成最后可执行程序时的各部件之间的联系及依赖关系。

makefile 的第一部分内容定义了一些变量，这些变量在 makefile 的后面部分

将会被替换掉。示例如下。

```
CC=             gcc
```

该行将 C 编译器定义为 gcc。然后在 makefile 的后面内容中，我们可以看到如下使用 CC 的例子。

```
diction:        diction.o sentence.o misc.o getopt.o getopt1.o
                $(CC) -o $@ $(LDFLAGS) diction.o sentence.o misc.o \
                getopt.o getopt1.o $(LIBS)
```

该例中包含了一个替代操作，在运行时$(CC)便被替换为 gcc。

大多数 makefile 文件都有很多行，这些行定义了目标文件（在本例中是 diction 可执行文件），也定义了目标文件所依赖的一些文件，剩下的行则描述的是那些将原文件生成目标文件的命令。就本例而言，我们可以看到可执行文件 diction 依赖于文件 diction.o、sentence.o、misc.o、getopt.o 以及 getopt1.o 的存在。继续查看 makefile 内容，我们可以看到这些目标文件的定义。

```
diction.o:      diction.c config.h getopt.h misc.h sentence.h
getopt.o:       getopt.c getopt.h getopt_int.h
getopt1.o:      getopt1.c getopt.h getopt_int.h
misc.o:         misc.c config.h misc.h
sentence.o:     sentence.c config.h misc.h sentence.h
style.o:        style.c config.h getopt.h misc.h sentence.h
```

然而，我们并没有看到为它们指定的任何命令。其实，它们是由文件前面的一个总体目标行生成的，该目标行描述了用来将所有.c 文件编译为.o 文件的命令。

```
.c.o:
                $(CC) -c $(CPPFLAGS) $(CFLAGS) $<
```

该命令行看起来确实复杂，那为什么不简单地列出编译的所有步骤然后照着步骤做呢？答案即将揭晓。揭晓前，我们先运行 make，来生成我们的程序。

```
[me@linuxbox diction-1.11]$ make
```

make 程序运行时，会使用 Makefile 文件中的内容指导其操作，其间会产生许多信息。

运行结束后，我们会看到所有的目标文件都出现在了目录中。

```
[me@linuxbox diction-1.11]$ ls
config.guess    de.po          en           install-sh    sentence.c
config.h        diction        en_GB        Makefile      sentence.h
config.h.in     diction.1      en_GB.mo     Makefile.in   sentence.o
config.log      diction.1.in   en_GB.po     misc.c        style
```

```
config.status    diction.c        getopt1.c       misc.h      style.1
config.sub       diction.o        getopt1.o       misc.o      style.1.in
configure        diction.pot      getopt.c        NEWS        style.c
configure.in     diction.spec     getopt.h        nl          style.o
COPYING          diction.spec.in getopt_int.h     nl.mo       test
de               diction.texi     getopt.o        nl.po
de.mo            diction.texi.in INSTALL          README
```

在这些文件中，我们也看到了 diction 和 style 程序，这是我们最初希望生成的。是不是应该庆祝一番，因为我们刚刚成功利用源代码编译了第一个可执行程序！

好吧，抑制住好奇心，再次运行 make 命令。

```
[me@linuxbox diction-1.11]$ make
make: Nothing to be done for 'all'.
```

仅仅出现了这样一个奇怪的信息。到底怎么回事？为什么它没有再次生成该程序呢？其实，这便是 make 的神奇之处。make 并不是简单盲目地重新生成所有东西，它只会生成那些需要生成的文件。在所有的目标文件已经存在的情况下，make 会判定源文件没有任何改动，也就不会进行任何操作。当然，可以删除其中某个目标文件，然后再次运行 make 以观察 make 的执行情况。

```
[me@linuxbox diction-1.11]$ rm getopt.o
[me@linuxbox diction-1.11]$ make
```

可以看到，make 重新生成了 getopt.o，并重新链接 diction 和 style 程序，因为它们都依赖于被删除的文件。这一特性同样指出了 make 的另一个用法——可以维护目标文件的更新。make 坚持一个原则，就是目标文件要比依赖文件新。这个特性有着重要的意义。因为程序员会经常更新部分源代码，然后使用 make 生成一个新版本。make 能够确保所有需要基于刚更新的代码而生成的程序都会生成。如果使用 touch 命令更新上例中某个源代码文件，就会得到下面的结果。

```
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me      me      37164 2009-03-05 06:14 diction
-rw-r--r-- 1 me      me      33125 2007-03-30 17:45 getopt.c
[me@linuxbox diction-1.11]$ touch getopt.c
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me      me      37164 2009-03-05 06:14 diction
-rw-r--r-- 1 me      me      33125 2009-03-05 06:23 getopt.c
[me@linuxbox diction-1.11]$ make
```

make 运行后，可以看到目标文件已经比依赖文件新了。

```
[me@linuxbox diction-1.11]$  ls -l diction getopt.c
-rwxr-xr-x 1 me      me      37164 2009-03-05 06:24 diction
-rw-r--r-- 1 me      me      33125 2009-03-05 06:23 getopt.c
```

make 能够智能地仅生成需要执行 building 操作的目标文件，这一能力让程序员获益不少。其所带来的时间节省效益，虽然对于一些小的项目而言并不是那么明显，但是对于大项目来说确是非常重要。要知道，Linux 内核（一个不断进行修改和完善的程序）可是包含几百万行的代码。

23.3.4 安装程序

打包好的源代码一般包含一个特殊的 make 目标程序，它便是 install。该目标程序将会在系统目录下安装最后生成的可执行程序。通常，会安装在目录 /usr/local/bin 下，该目录是本地主机上生成软件的常用安装目录。然而，对于普通用户，该目录通常是不可写的，所以必须转换成超级用户才可以运行安装。

```
[me@linuxbox diction-1.11]$ sudo make install
```

安装结束后，就可以查看该程序是否可以运行。

```
[me@linuxbox diction-1.11]$ which diction
/usr/local/bin/diction
[me@linuxbox diction-1.11]$ man diction
```

在安装完之后，便可以使用该应用该程序了。

23.4 本章结尾语

本章中我们介绍了三个简单的命令，它们是./configure、make 和 make install，它们可以用于建立多个源代码软件包。同样，我们还介绍了 make 在软件维护的过程中所扮演的重要角色。make 程序并不局限于源代码编译，它还可以用于所有需要维护“目标/依赖”间关系的任务。

第四部分
编写 shell 脚本

第24章 编写第一个 shell 脚本

在之前的章节中，我们已经学习了一系列的命令行工具。虽然这些工具可以解决很多计算问题，但是我们在使用它们时只能在命令行中一个一个手动输入。如果可以让 shell 完成更多工作，岂不是更好？当然可以。通过自行设计，将命令行组合成程序的方式，shell 就可以独立完成一系列复杂的任务。我们可以通过编写 shell 脚本方式来实现。

24.1 什么是 shell 脚本

最简单的解释是，shell 脚本是一个包含一系列命令的文件。shell 读取这个文件，然后执行这些命令，就好像这些命令是直接输入到命令行中一样。

shell 很独特，因为它既是一个强大的命令行接口，也是一个脚本语言解释器。我们将会看到，大多数能够在命令行中完成的工作都可以在脚本中完成，反之亦然。

我们已经讲解了许多 shell 特性，但我们关注的是哪些经常直接在命令行中使用的特性。shell 还提供了一些通常（但不总是）在编写程序时才使用的特性。

24.2 怎样写 shell 脚本

为了成功创建和运行一个 shell 脚本，我们需要做三件事情。

1. **编写脚本。** shell 脚本是普通的文本文件。所以我们需要一个文本编辑器来编辑它。最好的文本编辑器可以提供“语法高亮”功能，从而能够看到脚本元素彩色代码视图。“语法高亮”可以定位一些常见的错误。vim、gedit、kate 和许多其他的文本编辑器都是编写 shell 脚本的不错选择。
2. **使脚本可执行。** 系统相当严格，它不会将任何老式的文本文件当做程序。这样做有充足的理由！所以我们需要将脚本文件的权限设置为允许执行。
3. **将脚本放置在 shell 能够发现的位置。** 当没有显式指定路径名时，shell 会自动地寻找某些目录，来查找可执行文件。为了最大程度的方便，我们会将脚本放置在这些目录下。

24.2.1 脚本文件的格式

为了保持编程的传统，我们将创建一个“hello world”的程序，演示一个非常简单的脚本。启动文本编辑器并且输入以下脚本。

```
#!/bin/bash

# This is our first script.

echo 'Hello World!'
```

这个脚本的最后一行看起来非常熟悉，仅仅是一个 echo 命令加上一个字符串参数。第二行也很熟悉，看起来很像在很多配置文件中用到的注释行。就 shell 脚本中的注释来说，它们可以放置在一行的最后，如下所示。

```
echo 'Hello World!' # This is a comment too
```

文本行中，在“#”符号后面的所有内容会被忽略：

很很多命令一样，它也是在命令行中工作：

```
[me@linuxbox ~]$ echo 'Hello World!' # This is a comment too
Hello World!
```

尽管命令行中的注释没有用，但是他们也能起作用。

脚本的第一行看起来有点神奇。由于它以符号“#”开头，看起来像是注释，但是它应该具有一定的意义，所以它不仅仅是注释。实际上，这个“#!”字符序列是一种特殊的结构，称之为 shebang。shebang 用来告知操作系统，执行后面的脚本应该使用的解释器的名字。每一个 shell 脚本都应该将其作为第一行。

将这个脚本保存为 hello_world。

24.2.2 可执行权限

下一步要做的事情是让脚本可执行。使用 chmod 命令可以轻松做到：

```
[me@linuxbox ~]$ ls -l hello_world
-rw-r--r-- 1 me me 63 2012-03-07 10:10 hello_world
[me@linuxbox ~]$ chmod 755 hello_world
[me@linuxbox ~]$ ls -l hello_world
-rwxr-xr-x 1 me me 63 2012-03-07 10:10 hello_world
```

对于脚本，有两种常见的权限设置：权限为 755 的脚本，每个人都可以执行；而权限为 700 的脚本，则只有脚本所有人才能执行。注意，为了能够执行脚本，它必须是可读的。

24.2.3 脚本文件的位置

设置完权限之后，现在来执行脚本：

```
[me@linuxbox ~]$ ./hello_world
Hello World!
```

为了使脚本运行，我们必须显式指定脚本文件的路径。如果不这样做，我得到下面的结果：

```
[me@linuxbox ~]$ hello_world
bash: hello_world: command not found
```

为何会这样呢？是什么让脚本有别于程序呢？结果证明，什么都没有。脚本本身没有问题，问题在于脚本的位置。在第 11 章，我们讨论了 PATH 环境变量，以及它对系统搜索可执行程序方面的影响。如果没有显式指定路径，则系统在查找可一个执行程序时，需要搜索一系列目录。这就是当我们在命令行中输入 ls 时，系统知道要执行/bin/ls 的原因。/bin 目录是系统会自动搜索的一个目录。目录列表存放在名为 PATH 的环境变量中。这个 PATH 变量包含一个由冒号分隔开的待搜索目录的列表。我们可以查看 PATH 的内容：

```
[me@linuxbox ~]$ echo $PATH
```

```
/home/me/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr
/games
```

我们在这里看到了目录列表。如果脚本位于该列表中的任何一个目录中，问题就解决了。注意列表中的第一个目录/home/me/bin。大多数 Linux 发行版会配置 PATH 变量，使其包含用户主目录中的 bin 目录，允许用户执行他们自己的程序。如果我们创建了 bin 目录，并且将脚本放置在这个目录内，该脚本应该就会向其他程序那样开始运行。

```
[me@linuxbox ~]$ mkdir bin
[me@linuxbox ~]$ mv hello_world bin
[me@linuxbox ~]$ hello_world
Hello World!
```

如果 PATH 环境变量不包括这个 bin 目录，我们也可以轻松进行添加，方法是在.bashrc 文件中增加下面这行：

```
export PATH=~/bin:"$PATH"
```

修改完毕之后，它会在每一个新的终端会话中生效。为了将这一修改应用到当前的终端会话，则必须让 shell 重新读取.bashrc 文件。读取的方式如下。

```
[me@linuxbox ~]$ . .bashrc
```

这个“.”命令和 source 命令相同，是 shell 内置命令，用来读取一个指定的 shell 命令文件，并将其看做是像从键盘中输入的一样。

注意 在 Ubuntu 系统中，如果存在~/bin 目录，则当执行用户的.bashrc 文件时，Ubuntu 系统会自动将~/bin 目录添加到 PATH 变量中。所以，在 Ubuntu 操作系统中，如果创建了~/bin 目录然后退出并再重新登录，一切也会正常运行。

24.2.4 脚本的理想位置

~/bin 目录是一个存放个人使用脚本的理想位置。如果我们编写了一个系统上所有用户都可以使用的脚本，则该脚本的传统位置是/usr/local/bin。系统管理员使用的脚本通常放置在/usr/local/sbin。在大多数情况下，本地支持的软件，无论是脚本或者是编译好的程序，应该放置在/usr/local 目录下，而不是/bin 或是/usr/bin 目录下。这些目录都是由 Linux 文件系统层次结构标准指定的，只能包含由 Linxu 发行商所提供和维护的文件。

24.3　更多的格式诀窍

之所以严肃认真地编写脚本，其中一个目的是为维护提供便利。容易维护的脚本可以被它的作者或其他人员进行修改，以适应变化的要求。而让脚本易于阅读和理解是一种方便维护的方法。

24.3.1　长选项名

我们学习的很多命令都有短选项名和长选项名。例如，ls 命令有很多选项，它们既可以用长选项名表示，也可以用短选项名表示。例如：

```
[me@linuxbox ~]$ ls -ad
```

和

```
[me@linuxbox ~]$ ls --all --directory
```

这两个命令一样。为了减少输入，当在命令行中输入选项时，短选项更可取。但是在编写脚本时，长选项名可以提高可读性。

24.3.2　缩进和行连接

当使用长选项命令时，将命令扩展为好几行，可以提高命令的可读性。在第 17 章中，我们查看了一个特别长的 find 命令示例。

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 -exec chmod 0600
'{}' ';' \) -or \( -type d -not -perm 0700 -exec chmod 0700 '{}' ';' \)
```

乍一看，该命令有点难以理解。在脚本中，如果以如下方式编写，该命令就比较容易理解了。

```
find playground \
        \( \
                -type f \
                -not -perm 0600 \
                -exec chmod 0600 '{}' ';' \
        \) \
        -or \
        \( \
                -type d \
                -not -perm 0700 \
                -exec chmod 0700 '{}' ';' \
        \)
```

通过行连接符（反斜杠-回车符序列）和缩进，读者可以清楚地理解这个复杂命令的逻辑。该技术在命令行中也奏效，但是很少使用，原因就是在输入和编辑时会相当麻烦。脚本和命令行的一个区别是，脚本可以使用制表符来实现缩进，但在命令行中，Tab 键用来激活自动补齐功能。

为编写脚本而配置 vim

vim 文本编辑器有很多的配置项。有几个常用的选项为编写脚本提供了方便。

“:syntax on”用来打开“语法高亮”。打开这个选项后，在查看 shell 时，不同的 shell 语法元素会以不同的颜色显示。这对于识别某些编程错误很有帮助。它看起来也很酷。注意，为了使用这种功能，必须安装 vim 文本编辑器的完整版，并且编辑的文件必须含有 shebang 来标识这是一个 shell 脚本文件。如果无法设置“:syntax on”，可以试试“:set syntax=sh”。

“:set health”用来将搜索的结果高亮显示。比如我们查找单词“echo”，在开启该选项后，则所有的“echo”单词都会高亮显示。

“:set tabstop=4”用来设置 Tab 键造成的空格长度。默认值是 8 列。将这个值设置成 4，会让长文本行更容易适应屏幕。

“:set autoindent”用来开启自动缩进特性。这个选项会让 vim 对新的一行的缩进程度和上一行保持一致。对很多编程结构来说，这就加快了输入速度。要停止缩进，则可以按下 Ctrl-D。

通过把这些命令（不需要前面的冒号字符）添加到-/.vimrc 文件，则这些改变会永久生效。

24.5 本章结尾语

本章与脚本有关，我们介绍了如何编写脚本，以及如何轻松地在系统中执行脚本。我们还介绍了如何使用各种格式技术来提升脚本的可读性（以及可维护性）。在随后的章节中，易于维护将作为编写良好脚本的核心原则多次出现。

第25章 启动一个项目

从本章开始，我们将开始构建一个程序。该项目的目的是查看如何使用各种 shell 特性来创建程序，而且更为重要的是，如何创建好的程序。

我们将要编写的程序是一个报告生成器，它会显示系统的各种统计数据和状态。并以 HTML 格式来产生该报告。这样，我们就可以使用 Web 浏览器进行查看。

程序一般由一系列阶段组成，每一个阶段都会增加一些特性和功能。该程序的第一个阶段是创建一个非常小的 HTML 页面，它不包含任何系统信息（后面会添加这些信息）。

25.1 第一阶段：最小的文档

我们需要知道的第一件事就是一个结构良好的 HTML 文档的格式，如下所示。

```
<HTML>
        <HEAD>
                <TITLE>Page Title</TITLE>
```

```
        </HEAD>
        <BODY>
                Page body.
        </BODY>
</HTML>
```

如果将这些内容输入到文本编辑器，并将该文件保存为 foo.html，就可以在火狐浏览器中输入“file:///home/username/foo.html”这个 URL 地址查看该文件。

程序的第一个阶段是将这个 HTML 文件输出到标准输出。我们可以编写一个程序，来很容易地完成该任务。启动文本编辑器，然后创建一个名为“~/bin/sys_info_page”的新文件。

```
[me@linuxbox ~]$ vim ~/bin/sys_info_page
```

随后输入以下的程序。

```
#!/bin/bash

# Program to output a system information page

echo "<HTML>"
echo "   <HEAD>"
echo "               <TITLE>Page Title</TITLE>"
echo "   </HEAD>"
echo "   <BODY>"
echo "               Page body."
echo "   </BODY>"
echo "</HTML>"
```

我们第一次要尝试解决的这个程序包含了一个“shebang”、一条“注释”(总是一个比较好的习惯）和一系列 echo 命令。每一个 echo 命令用来显示一行。保存该文件之后，就其成为可执行文件，并尝试去运行它。

```
[me@linuxbox ~]$ chmod 755 ~/bin/sys_info_page
[me@linuxbox ~]$ sys_info_page
```

该程序在运行时，我们就可以看到这个 HTML 文档的文本显示在屏幕上，这是因为脚本中的 echo 命令将其输出发送到了标准输出。再一次运行该程序，并且将该程序的输出重新定向到 sys_info_page.html 文件，这样就可以用 Web 浏览器查看结果了。

```
[me@linuxbox ~]$ sys_info_page > sys_info_page.html
[me@linuxbox ~]$ firefox sys_info_page.html
```

目前为止，一切顺利。

在编写程序时，最好尽力保持程序的清晰明了。当一个程序利于阅读和理解时，维护起来也会很容易，而且通过减少输入量，也会程序更容易编写。该

程序的当前版本虽然工作良好，但是可以更简洁一些。实际上，我们可以将所有的 echo 命令整合为一个命令，这样就可以更容易地将更多的行添加到程序的输出中。现在，将程序修改为如下所示：

```
#!/bin/bash

# Program to output a system information page

echo "<HTML>
        <HEAD>
                <TITLE>Page Title</TITLE>
        </HEAD>
        <BODY>
                Page body.
        </BODY>
</HTML>"
```

一个带引号的字符串可以包含换行符，因此也就可以包含多个文本行。shell 将持续读取文本，直到读取到下一个引号为止。它在命令行中也是这样工作的：

```
[me@linuxbox ~]$ echo "<HTML>
>         <HEAD>
>                 <TITLE>Page Title</TITLE>
>         </HEAD>
>         <BODY>
>                 Page body.
>         </BODY>
> </HTML>"
```

每行开头的“>”字符是包含在 PS2 shell 变量中的 shell 提示符。每当我们在 shell 中输入多行语句时就会出现。现在来看，这个功能现在还不是很明显，但当我们介绍多行编程语句时，它会相当方便。

25.2 第二阶段：加入一点数据

现在该程序可以生成一个最小的文档，我们在这个文档中加入一些数据。为了实现这个目标，需要做以下改动。

```
#!/bin/bash

# Program to output a system information page

echo "<HTML>
        <HEAD>
                <TITLE>System Information Report</TITLE>
        </HEAD>
        <BODY>
                <H1>System Information Report</H1>
        </BODY>
</HTML>"
```

我们加入了一个页面标题，并为报告正文部分添加了一个标题。

25.3 变量和常量

但是，脚本现在还有一个问题。请注意字符串“System Information Report”是如何被重复的。对于我们这个小脚本来说，这不是问题。但是，如果我们的脚本很长，而且多处都存在该字符串，现在我们想将标题修改为其他内容，则需要在多个地方进行修改，这样工作量就大了。我们是否可以修改脚本，使得字符串只出现一次而不是多次呢？而且这也会让脚本在日后的维护更加简单。为此，可以这样做：

```
#!/bin/bash

# Program to output a system information page

title="System Information Report"

echo "<HTML>
        <HEAD>
                <TITLE>$title</TITLE>
        </HEAD>
        <BODY>
                <H1>$title</H1>
        </BODY>
</HTML>"
```

通过创建一个叫名为 title 的变量，并且将其赋值为“System Information Report”，就可以利用参数扩展能，将该字符串放置到多个地方了。

25.3.1 创建变量和常量

如何创建变量呢？这很简单，我们刚刚使用过。当 shell 遇到一个变量时，会自动创建这个变量。这点与大多数程序中，使用一个变量时必须先声明或定义有所不同。在这个方面，shell 非常的宽松，但也会导致一些问题。例如，请看以下命令行中出现的场景。

```
[me@linuxbox ~]$ foo="yes"
[me@linuxbox ~]$ echo $foo
yes
[me@linuxbox ~]$ echo $fool
[me@linuxbox ~]$
```

首先我们为变量 foo 赋值 yes，再使用 echo 命令显示 foo 的值。然后我们显示一个由拼写错误造成的变量 fool 的值，而得到了一个空值。这是因为 shell 在

遇到 fool 变量时，会轻松地创建这个变量，并且为这个 fool 变量赋了一个默认的空值。从这点可以看出，我们必须对拼写有足够的关注。同进，也要理解在这个示例中，究竟发生了什么。我们前面学到，shell 会执行扩展，所以命令

```
[me@linuxbox ~]$ echo $foo
```

会经历参数扩展，而且结果是:

```
[me@linuxbox ~]$ echo yes
```

然而命令

```
[me@linuxbox ~]$ echo $fool
```

则扩展为

```
[me@linuxbox ~]$ echo
```

这个为空的变量在扩展后为空。这对需要使用参数的命令说来，将会引起混乱。例如：

```
[me@linuxbox ~]$ foo=foo.txt
[me@linuxbox ~]$ foo1=foo1.txt
[me@linuxbox ~]$ cp $foo $fool
cp: missing destination file operand after 'foo.txt'
Try 'cp --help' for more information.
```

我们为变量 foo 和 fool 赋值，然后执行 cp 命令，但将第二个参数名拼错了。在参数扩展后，这个 cp 命令仅接受了一个参数，但是它需要的是两个。

变量的命名规则如下所示。

- 变量名称应由字母、数字和下划线组成。
- 变量名称的第一个字符必须是字母或者下划线。
- 变量名称中不允许空格和标点。

变量意味着值会发生变化，并且在大多数应用中，变量就是这样使用的。在前面的应用中，变量 title 其实是作为常量使用的。常量其实就是一个有名称和确定值的变量。区别就是，常量的值不会发生变化。在一个执行几何计算的应用中，我们可以定义 PI 为一个常量，并且将它的值赋为 3.1415，从而无需在程序中使用数值 3.1415。对于 shell 来说，变量和常量没有区别，这种划分更多的是为了编程者的方便。较为普遍的约定是，我们使用大写字母表示常量，使用小写字母表示变量。我们将前面的 shell 按照这个约定进行修改。

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"

echo "<HTML>
        <HEAD>
                <TITLE>$TITLE</TITLE>
        </HEAD>
        <BODY>
                <H1>$TITLE</H1>
        </BODY>
</HTML>"
```

利用这个机会，我们为 shell 中的变量 HOSTNAME 赋值。这个 HOSTNAME 是机器在网络上的名称。

注意　实际上，shell 还提供了强制常量不发生变化的方法，这是通过使用带有-r 选项(只读)的 delcare 内置命令实现的。我们在这里使用这种方式为变量 TITLE 赋值：

```
declare -r TITLE= "Page Title"
```

shell 将阻止一切后续的向 TITLE 的赋值。这个特性很少使用，但在很早的脚本中曾经存在。

25.3.2 为变量和常量赋值

这是我们开始真正使用扩展的地方。我们已经看到，变量用以下方式来赋值：

```
variable=value
```

其中，variable 是变量的名称，value 是变量的值。和大多数的编程语言不同，shell 并不关心赋给变量的值数值类型，它会将其都当成字符串。通过使用带-i 选项的 declare 命令，可以强制 shell 将变量限制为整型数值。但是，这如同将变量设置成只读模式一样，我们很少这样做。

注意，在赋值时，变量名、等号和值之间不能含有空格。那么，变量的值中可以包括什么呢？它包含的是可以扩展为字符串的任意值：

```
a=z                         #将字符串"z"赋值给变量 a
b= "a string"               #嵌入的空格必须用引号括起来
c= "a string and $b"            #可以被扩展到赋值语句中的其他扩展，比如变量
d=$(ls-l foo.txt)               #命令的结果
e=$((5*7))                  #算术扩展
f= "\t\ta string\n"             #转义序列，比如制表符和换行符
```

可以在一行中给多个变量赋值。

```
a=5 b="a string"
```

在扩展期间，变量名称可以用花括号“{}”括起来。当变量名因为周围的上下文而变得不明确时，这就会很有帮助了。在这里，我们使用变量将一个文件的名字由 myfile 改为 myfile1：

```
[me@linuxbox ~]$ filename="myfile"
[me@linuxbox ~]$ touch $filename
[me@linuxbox ~]$ mv $filename $filename1
mv: missing destination file operand after 'myfile'
Try 'mv --help' for more information.
```

因为 shell 将 mv 命令的第二个参数当成了一个新的变量，所以这样做没有成功。该问题可以用以下方法解决。

```
[me@linuxbox ~]$ mv $filename ${filename}1
```

用花括号括起来后，shell 将不会把跟在后面的“1”认为是变量名称的一部分。

我们现在添加一些数据到我们的报告中，比如报告创建的日期、时间，以及报告创建者的用户名。

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

echo "<HTML>
        <HEAD>
                <TITLE>$TITLE</TITLE>
        </HEAD>
        <BODY>
                <H1>$TITLE</H1>
                <P>$TIME_STAMP</P>
        </BODY>
</HTML>"
```

25.4 here 文档

我们已经了解了两种通过 echo 命令输出文本的不同方法。此外还有称之为“hure 文档”或“here 脚本”的第三种方法来输出文本。here 文档是 I/O 重定向

的另外一种形式，我们在脚本中嵌入正文文本，然后将其输入到一个命令的标准输入中。工作方式如下：

```
command << token
text
token
```

其中，command 是接受标准输入的命令名，token 是用来指示嵌入文本结尾的字符串。现在修改脚本，使其使用 here 文档。

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

cat << _EOF_
<HTML>
        <HEAD>
                <TITLE>$TITLE</TITLE>
        </HEAD>
        <BODY>
                <H1>$TITLE</H1>
                <P>$TIME_STAMP</P>
        </BODY>
</HTML>
_EOF_
```

我们的脚本不再使用 echo，而是使用 cat 和 here 文档。字符串_EOF_”（含义是“文件结尾”）被选作 token，并且指示嵌入文本的结尾。注意，token 必须在一行中单独出现，而且文本行的末尾没有空格。

那么使用 here 文档有什么好处呢？它和 echo 命令很类似，但是在默认情况下，here 文档内的单引号和双引号将失去它们在在 shell 中的特殊含义。下面是一个命令行示例。

```
[me@linuxbox ~]$ foo="some text"
[me@linuxbox ~]$ cat << _EOF_
> $foo
> "$foo"
> '$foo'
> \$foo
> _EOF_
some text
"some text"
'some text'
$foo
```

可以看到，shell 没有注意到引号，而是将引号当做普通字符。这就可以在

here 文档中随意嵌入引号。这给我们的报告程序带来了方便。

here 文档可以和能够接受标准输入的任何命令一起使用。在本例中，我们使用 here 文档向 ftp 程序传递一系列命令，以从远程 FTP 服务器上获取一个文件。

```
#!/bin/bash

# Script to retrieve a file via FTP

FTP_SERVER=ftp.nl.debian.org
FTP_PATH=/debian/dists/lenny/main/installer-i386/current/images/cdrom
REMOTE_FILE=debian-cd_info.tar.gz

ftp -n << _EOF_
open $FTP_SERVER
user anonymous me@linuxbox
cd $FTP_PATH
hash
get $REMOTE_FILE
bye
_EOF_
ls -l $REMOTE_FILE
```

如果将重定向操作符由“<<”改为“<<-”，shell 就会忽略在 here 文档中开图的 Tab 字符。这样就能缩进 here 文档，从而提供可读性。

```
#!/bin/bash

# Script to retrieve a file via FTP

FTP_SERVER=ftp.nl.debian.org
FTP_PATH=/debian/dists/lenny/main/installer-i386/current/images/cdrom
REMOTE_FILE=debian-cd_info.tar.gz

ftp -n <<- _EOF_
	open $FTP_SERVER
	user anonymous me@linuxbox
	cd $FTP_PATH
	hash
	get $REMOTE_FILE
	bye
	_EOF_
ls -l $REMOTE_FILE
```

25.5 本章结尾语

在本章，我们启动了一个项目，该项目带领我们经历了构建一个成功脚本的整个过程。我们还介绍了变量和常量的概念，以及使用它们的方式。它们是第一批使用参数扩展的应用程序。我们还查看了如何从脚本中产生输出，以及嵌入文本块的各种方法。

第26章

自顶向下设计

随着程序越来越大，越来越复杂，设计、编码和维护都将越来越困难。所以，在设计任何大型项目时，我们最好将庞大的、复杂的任务，拆分成一系列小的、简单的任务。

想象一下，我们来描述一个常见的日常任：一个火星人去市场买食物。我们可以按照如下步骤来描述整个过程。

1. 上车。
2. 开往市场。
3. 停车。
4. 进入市场。
5. 购买食物。
6. 回到车里。

7．开车回家。

8．停车。

9．进屋。

但是，这个火星人需要更多细节。我们可以将子任务“停车”进一步拆分为一系列步骤。

1．找停车的位置。

2．将车开入停车位置。

3．关闭发动机。

4．拉手刹。

5．下车。

6．锁上车门。

“关闭发动机”子任务又可以进一步的分解成包括“熄火”和“拔出点火钥匙”等步骤。这样逐步分解，直到全部定义了“去市场”这个过程的整个步骤。

这种先确定上层步骤，然后再逐步细化这些步骤的过程，称为自顶向下设计。

通过这种设计方式，可以将大而复杂的任务分解成很多小而简单的任务。自顶向下设计是一种设计程序的常见方式，尤其适合 shell 编程。

本章我们将会使用自顶向下设计方式，进一步开发我们的报告生成器脚本。

26.1 shell 函数

我们的报告当前执行如下步骤，即可生成 HTML 文档。

1．打开页面。

2．打开页面标题。

3．设置页面标题。

4．关闭页面标题。

5．打开页面主体。

6．输出页面主体。

7. 输出时间戳。

8. 关闭页面主题。

9. 关闭页面。

为了下一阶段的开发，我们将在步骤 7 和步骤 8 之间额外添加一些任务，如下所示。

- **系统正常运行时间和负载**。这是自上次关机或重启之后系统的运行时间，在几个时间间隔内，当前运行在处理器上的平均任务量。
- **磁盘空间**。系统存储空间的使用情况。
- **用户空间**。每个用户所使用的存储空间。

在这些任务中，如果每个任务都有一条对应的命令，我们可以直接通过命令替换的方式，将它们添加到脚本中。

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"

CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

cat << _EOF_
<HTML>
        <HEAD>
                <TITLE>$TITLE</TITLE>
        </HEAD>
        <BODY>
                <H1>$TITLE</H1>
                <P>$TIME_STAMP</P>
                $(report_uptime)
                $(report_disk_space)
                $(report_home_space)
        </BODY>
</HTML>
_EOF_
```

我们可以通过两种方式创建这些额外的命令。我们可以编写 3 个独立的脚本，并把它们放置到环境变量 PATH 所列出的目录中，或者是我们可以将脚本作为 shell 函数嵌入到程序中。前面提到，shell 函数是位于其他脚本中的迷你脚本，可以用作自主程序（autonomous program）。shell 函数有两种语法形式。第一种如下所示：

```
function name {
         commands
         return
}
```

其中 *name* 是指这个函数的名称，*commands* 是这个函数中的一系列命令。第二种看起来如下所示：

```
name () {
        commands
        return
}
```

这两种格式等价，并且可以交替使用。来看下面这个脚本，它演示了 shell 函数的使用。

```
1     #!/bin/bash
2
3     # Shell function demo
4
5     function funct {
6             echo "Step 2"
7             return
8     }
9
10    # Main program starts here
11
12    echo "Step 1"
13    funct
14    echo "Step 3"
```

当 shell 读取脚本的时候，它会跳过第 1～11 行，这些行包含的是注释和函数的定义。执行从带有 echo 命令的第 12 行开始。第 13 行调用了 shell 函数 funct，shell 会执行这个函数，而且其执行方式与执行其他命令时相同。程序控制权然后移动到第 6 行，执行第 2 个 echo 命令，随后再执行第 7 行。这个 return 命令终止函数的执行，然后将控制权还给函数调用后面的代码（第 14 行），然后执行最后一个 echo 命令。注意，为了让函数调用被识别为 shell 函数，而不是被解释为外部程序的名字，shell 函数的定义义在脚本中的位置必须在它被调用的前面。

我们在脚本中添加上最小的 shell 函数定义：

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

report_uptime () {
        return
}
report_disk_space () {
        return
}
report_home_space () {
```

```
        return
}

cat << _EOF_
<HTML>
        <HEAD>
                <TITLE>$TITLE</TITLE>
        </HEAD>
        <BODY>
                <H1>$TITLE</H1>
                <P>$TIME_STAMP</P>
                $(report_uptime)
                $(report_disk_space)
                $(report_home_space)
        </BODY>
</HTML>
_EOF_
```

shell 函数的命名规则和变量相同。一个函数必须至少包含一条命令。return 命令（可选的）可以满足该要求。

26.2 局部变量

在目前我们所编写的脚本中，所有的变量（包括常量）都是全局变量。全局变量在整个程序期间会一直存在。在很多情况下，这都不错。但是有时候，它会让 shell 函数的使用变得复杂。在 shell 函数中，经常需要的是局部变量。局部变量仅仅在定义它们的 shell 函数中有效，一旦 shell 函数终止，它们就不再存在。

局部变量可以让程序员使用已经存在的变量名称，无论是脚本中的全局变量，还是其他 shell 函数中的变量，而不用考虑潜在的命名冲突。

下面是一个显示如何定义和使用局部变量的脚本示例。

```
#!/bin/bash

# local-vars: script to demonstrate local variables

foo=0 # global variable foo

funct_1 () {

        local foo # variable foo local to funct_1

        foo=1
        echo "funct_1: foo = $foo"
}
funct_2 () {
        local foo # variable foo local to funct_2

        foo=2
        echo "funct_2: foo = $foo"
}
```

```
echo "global: foo = $foo"
funct_1
echo "global: foo = $foo"
funct_2
echo "global: foo = $foo"
```

可以看到，局部变量是通过在变量名前面添加单词 local 来定义的。这样，就创建并同时定义了一个 shell 函数中的局部变量。一旦出了这个 shell 函数，这个局部变量将不再存在。当运行该脚本时，结果如下所示：

```
[me@linuxbox ~]$ local-vars
global: foo = 0
funct_1: foo = 1
global: foo = 0
funct_2: foo = 2
global: foo = 0
```

可以看到，在两个 shell 函数中对局部变量 foo 的赋值，不影响 shell 函数以外定义的变量 foo 的值。

这个特性可以让我们编写的 shell 函数相互独立，而且也独立于它们所在的脚本。这非常有用，因为它有助于阻止程序中各个部分的相互干扰。该特性也可以让我们编写出可移植的 shell 函数。也就是说，我们可以根据需要将某一个脚本中的 shell 函数剪切下来，然后粘贴到另外一个脚本中。

26.3 保持脚本的运行

在开发程序时，让程序保持可运行的状态会非常有用。通过这种方式，并经常测试，就可以在开发过程的早期检测到错误。这也会让问题的调试变得容易。例如，如果我们运行某个程序，并对它做了细微改动，然后再次运行该程序，此时发现了问题。这可能有可能是最近的改动导致的。通过添加空函数（程序员将其称为 stub），我们可以在早期验证程序的逻辑流程。当构建一个 stub 时，最好是包含一些能够为程序员提供反馈信息的东西，这些反馈信息可以显示正在执行的逻辑流程。如果现在查看脚本的输出，会发现在时间戳之后的输出中有一些空行，但是我们不确定是什么原因导致的。

```
[me@linuxbox ~]$ sys_info_page
<HTML>
        <HEAD>
                <TITLE>System Information Report For twin2</TITLE>
        </HEAD>
        <BODY>
                <H1>System Information Report For linuxbox</H1>
```

```
                <P>Generated 03/19/2012 04:02:10 PM EDT, by me</P>

        </BODY>
</HTML>
```

我们修改该函数，使其包含一些反馈信息。

```
report_uptime () {
        echo "Function report_uptime executed."
        return
}

report_disk_space () {
        echo "Function report_disk_space executed."
        return
}

report_home_space () {
        echo "Function report_home_space executed."
        return
}
```

然后再次运行该脚本。

```
[me@linuxbox ~]$ sys_info_page
<HTML>
        <HEAD>
                <TITLE>System Information Report For linuxbox</TITLE>
        </HEAD>
        <BODY>
                <H1>System Information Report For linuxbox</H1>
                <P>Generated 03/20/2012 05:17:26 AM EDT, by me</P>
                Function report_uptime executed.
                Function report_disk_space executed.
                Function report_home_space executed.
        </BODY>
</HTML>
```

这次可以发现，我们的三个函数事实上都执行了。

由于我们的函数框架没有问题，因此，需要更新一些函数代码。首先是 report_uptime 函数。

```
report_uptime () {
        cat <<- _EOF_
                <H2>System Uptime</H2>
                <PRE>$(uptime)</PRE>
                EOF_
        return
}
```

该函数的代码直截了当。我们使用了一个 here 文档来输出一个标题，并输出 uptime 命令的结果。命令结果使用<PRE>标签围起来，其目的是保留命令的格式。report_disk_space 函数也类似。

```
report_disk_space () {
        cat <<- _EOF_
                <H2>Disk Space Utilization</H2>
                <PRE>$(df -h)</PRE>
                _EOF_
        return
}
```

这个函数使用了 df-h 命令来检查磁盘的空间。随后，我们来构建 report_home_space 函数。

```
report_home_space () {
        cat <<- _EOF_
                <H2>Home Space Utilization</H2>
                <PRE>$(du -sh /home/*)</PRE>
                _EOF_
        return
}
```

我们使用带-sh 选项的 du 命令来执行这个任务。这个并不是问题的完整解决方案。虽然它可以在某些操作系统中（例如 Ubuntu）运行，但是在其他系统中则无效。这是因为很多系统设置了主目录的权限，以防止被其他用户读取，这种安全措施也很合理。在这些操作系统中，只有在超级用户运行我们的脚本时，我们编写的这个 report_home_space 函数才会执行。一个更好的解决方案是让脚本根据用户的权限调整自己的行为。我们在下一章讨论该问题。

你的.bashrc 文件中的 shell 函数

shell 函数可以很好地取代别名，并且实际上也是创建个人使用的小命令的首选方法。别名非常局限于命令的种类和它们支持的 shell 特性，而 shell 特性则允许任何可以编写为脚本的东西。例如，如果我们很喜欢为脚本开发的 report_disk_space 这个 shell 函数，则可以为我们的.bashrc 文件创建一个相似的名为 ds 的函数。

```
ds () {
    echo "Disk Space Utilization For $HOSTNAME"
    df -h
}
```

26.4 本章结尾语

在本章，我们介绍了一种常用的一种程序方式——自顶向下设计，并知道了如何使用 shell 函数按照要求来构建逐步细化的任务。我们还学习了如何使用局部变量时 shell 函数相互独立，并独立于它们所在的程序。这样，我们就可以以可移植和可重用的方式编写 shell 函数，并将其用到多个程序中，从而节省大量的时间。

第27章

流控制：IF 分支语句

上一章中，我们提出了一个问题。如何使我们的报告生成器脚本能适应运行该脚本的用户的权限？该问题的解决方案要求我们能找到一种方法，在脚本中能基于测试的结果来“改变方向”。用编程术语来说，就是我们需要程序能够分支。

让我们来考虑一个使用伪代码表示的简单逻辑示例。伪代码是计算机语言的一种模拟，为的是方便人们理解。

```
X = 5
If X = 5, then:
        Say "X equals 5."
Otherwise:
        Say "X is not equal to 5."
```

这就是一个分支的例子。根据条件，如果“X=5”，表示为“X 等于 5”；否则，则说“X 不等于 5”。

27.1 使用 if

通过 shell，我们可以对上面的逻辑进行编码，如下所示：

```
x=5

if [ $x = 5 ]; then
	echo "x equals 5."
else
	echo "x does not equal 5."
fi
```

或者可以直接在命令行中输入以上代码（略有简化）。

```
[me@linuxbox ~]$ x=5
[me@linuxbox ~]$ if [ $x = 5 ]; then echo "equals 5"; else echo "does not equal
5"; fi
equals 5
[me@linuxbox ~]$ x=0
[me@linuxbox ~]$ if [ $x = 5 ]; then echo "equals 5"; else echo "does not equal
5"; fi
does not equal 5
```

在这个例子中，我们执行了两次命令。第一次，设置变量 X 为 5，从而输出字符串“equals 5”；第二次，设置变量X 为 0，从而输出字符串“does not equal 5”。

if 语句的语法格式如下。

```
if commands; then
	commands
[elif commands; then
	commands...]
[else
	commands]
fi
```

在这个语法格式中，“command”可以是一组命令。乍看上去可能会有些迷惑。在去除这个迷惑前，我们必须先解一下 shell 如何判断一个命令的成功与失败的。

27.2 退出状态

命令（包括我们编写的脚本和 shell 函数）在执行完毕后，会向操作系统发送一个值，称之为“退出状态”。这个值是一个 0～255 的整数，用来指示命令执行成功还是失败。按照惯例，数值 0 表示执行成功，其他的数值表示

执行失败。shell 提供了一个可以用来检测退出状态的参数。在下面的例子中可以看到。

```
[me@linuxbox ~]$ ls -d /usr/bin
/usr/bin
[me@linuxbox ~]$ echo $?
0
[me@linuxbox ~]$ ls -d /bin/usr
ls: cannot access /bin/usr: No such file or directory
[me@linuxbox ~]$ echo $?
2
```

在这个例子中，我们两次执行了 ls 命令。第一次，命令执行成功，如果显示参数“$?”的值，可以看到它是 0。第二次执行 ls 命令时，产生了一个错误，再次显示参数“$?”的值，这次则为 2，表示这个命令遇到了一个错误。有些命令使用不同的退出值来诊断错误，而许多命令在执行失败时，只是简单地退出并发送数字 1。man 手册中经常会包括一个标题为“Exit Status”的段落，它描述使用的代码。数字 0 总是表示执行成功。

shell 提供了两个非常简单的内置命令，它们不做任何事情，除了以一个 0 或 1 退出状态来终止执行。“true”命令总是表示执行成功，而“false”命令总是表示执行失败。

```
[me@linuxbox ~]$ true
[me@linuxbox ~]$ echo $?
0
[me@linuxbox ~]$ false
[me@linuxbox ~]$ echo $?
1
```

我们可以用这两个命令来查看 if 语句是如何工作的。if 语句真正做的事情是评估命令的成功或失败。

```
[me@linuxbox ~]$ if true; then echo "It's true."; fi
It's true.
[me@linuxbox ~]$ if false; then echo "It's true."; fi
[me@linuxbox ~]$
```

当在 if 后面的命令执行成功时，命令 echo "It's true."会被执行，而当在 if 后面的命令执行失败时，该命令则不执行。如果在 if 后面有一系列的命令，那么则根据最后一个命令的执行结果进行评估。

```
[me@linuxbox ~]$ if false; true; then echo "It's true."; fi
It's true.
[me@linuxbox ~]$ if true; false; then echo "It's true."; fi
[me@linuxbox ~]$
```

27.3 使用 test 命令

目前为止，经常和 if 一起使用的命令是 test。test 命令会执行各种检查和比较。这个命令有两种等价的形式：

```
test expression
```

以及更流行的：

```
[ expression ]
```

这里 expression 是一个表达式，其结果是 true 或 false。当这个表达式为 true 时，test 命令返回一个零退出状态；当表达式为 false 时，test 命令的退出状态为 1。

27.3.1 文件表达式

表 27-1 中的表达式用来评估文件的状态。

表 27-1 测试文件的表达式

表达式	成为 true 的条件
file1 –ef file2	file1 和 file2 拥有相同的信息节点编号（这两个文件通过硬链接指向同一个文件）
file1 –nt file2	file1 比 file 2 新
file1 –ot file2	file 1 比 file 2 旧
-b file	file 存在并且是一个块（设备）文件
-c file	file 存在并且是一个字符（设备）文件
-d file	file 存在并且是一个目录
-e file	file 存在
-f file	file 存在并且是一个普通文件
-g file	file 存在并且设置了组 ID
-G file	file 存在并且属于有效组 ID
-k file	file 存在并且有“粘滞位（sticky bit）”属性
-L file	file 存在并且是一个符号链接
-O file	file 存在并且属于有效用户 ID
-p file	file 存在并且是一个命名管道
-r file	file 存在并且可读（有效用户有可读权限）
-s file	file 存在并且其长度大于 0
-S file	file 存在并且是一个网络套接字

续表

表达式	成为 true 的条件
-t fd	fd 是一个定向到终端/从终端定向的文件描述符，可以用来确定标准输入/输出/错误是否被重定向
-u file	file 存在并且设置了 setuid 位
-w file	file 存在并且可写（有效用户拥有可写权限）
-x file	fiile 存在并且可执行（有效用户拥有执行/搜索权限）

下面的脚本可用来演示某些文件表达式。

```
#!/bin/bash

# test-file: Evaluate the status of a file

FILE=~/.bashrc

if [ -e "$FILE" ]; then
        if [ -f "$FILE" ]; then
                echo "$FILE is a regular file."
        fi
        if [ -d "$FILE" ]; then
                echo "$FILE is a directory."
        fi
        if [ -r "$FILE" ]; then
                echo "$FILE is readable."
        fi
        if [ -w "$FILE" ]; then
                echo "$FILE is writable."
        fi
        if [ -x "$FILE" ]; then
                echo "$FILE is executable/searchable."
        fi
else
        echo "$FILE does not exist"
        exit 1
fi

exit
```

这个脚本会评估赋值给常量 FILE 的文件，并显示评估结果。关于该脚本，需要注意两个有趣的地方。首先，要注意$FILE 在表达式内是怎样被引用的。尽管引号不是必需的，但是这可以防范参数为空的情况。如果$FILE 的参数扩展产生一个空值，将导致一个错误（操作符会被解释为非空的字符串，而不是操作符）。用引号把参数括起来可以确保操作符后面总是跟随着一个字符串，即使字符串为空。其次，注意脚本末尾的 exit 命令。这个 exit 命令接受一个单独的可选参数，它将成为脚本的退出状态。当不传递参数时，退出状态默认为 0。以这种方法使用 exit 命令，当$FILE 扩展为一个不存在的文件名时，可以允许

脚本提示失败。这个 exit 命令出现在脚本的最后一行。这样，当脚本执行到最后时，不管怎样，默认情况下它将以退出状态零终止。

类似地，通过在 return 命令中包含一个整数参数，shell 函数可以返回一个退出状态。如果要将上面的脚本转换为一个 shell 函数，从而能够在一个更大的程序中使用，可以将 exit 命令替换为 return 命令，并得到想要的行为。

```
test_file () {

        # test-file: Evaluate the status of a file

        FILE=~/.bashrc

        if [ -e "$FILE" ]; then
                if [ -f "$FILE" ]; then
                        echo "$FILE is a regular file."
                fi
                if [ -d "$FILE" ]; then
                        echo "$FILE is a directory."
                fi
                if [ -r "$FILE" ]; then
                        echo "$FILE is readable."
                fi
                if [ -w "$FILE" ]; then
                        echo "$FILE is writable."
                fi
                if [ -x "$FILE" ]; then
                        echo "$FILE is executable/searchable."
                fi
        else
                echo "$FILE does not exist"
                return 1
        fi
}
```

27.3.2 字符串表达式

表 27-2 中的表达式用来测试字符串的操作。

表 27-2 测试字符串表达式

表达式	成为 true 的条件
string	string 不为空
-n string	string 的长度大于 0
-z string	string 的长度等于 0
string1=string2 string1==string2	string1 和 string2 相等。单等号和双等号都可以使用，但是双等号使用的更多
string1!=string2	string1 和 string2 不相等

续表

表达式	成为 true 的条件
string1>string2	在排序时，string1 在 string2 之后
string1<string2	在排序时，string1 在 string2 之前

警告 在使用 test 命令时，“>” 和 “<” 运算符必须用引号括起来（或者是使用反斜杠进行转义）。如果不这样做，就会被 shell 解释为重定向操作符，从而造成潜在的破坏性结果。同时注意，尽管 bash 文档中已经声明，排序遵从当前语系的排列规则，但并非如此。在 bash 4.0 版本以前（包括 4.0 版本），使用的是 ASCII（POSIX）排序方式。

下面是一个合并字符串表达式的脚本。

```
#!/bin/bash

# test-string: evaluate the value of a string

ANSWER=maybe

if [ -z "$ANSWER" ]; then
        echo "There is no answer." >&2
        exit 1
fi
if [ "$ANSWER" = "yes" ]; then
        echo "The answer is YES."
elif [ "$ANSWER" = "no" ]; then
        echo "The answer is NO."
elif [ "$ANSWER" = "maybe" ]; then
        echo "The answer is MAYBE."
else
        echo "The answer is UNKNOWN."
fi
```

在这个脚本中，我们评估了常量 ANSWER。我们首先检查了这个字符串是否是空。如果是空，则终止脚本，并且把退出状态设置为 1。注意应用到 echo 命令的重定向操作。它把错误消息 “There is no answer.” 重定向到标准错误，这也是处理错误信息的 “合理” 方法。如果字符串非空，则评估这个字符串的值是否是 “yes”、“no” 或者 “maybe”。我们使用 elif（else if 的缩写）来实现上述目的。通过使用 elif，我们可以建立一个更复杂的逻辑测试。

27.3.3 整数表达式

表 27-3 中的表达式用于整数。

表 27-3 整数判断操作

表达式	成为 true 的条件
integer1 –eq integer2	integer1 和 integer2 相等
integer1 –ne integer2	integer1 和 integer2 不相等
integer1 –le integer2	integer1 小于等于 integer2
integer1 –lt integer2	integer1 小于 integer2
integer1 –ge integer2	integer1 大于等于 integer2
integer1 –gt integer2	integer1 大于 integer2

下面是一个演示这些表达式的脚本。

```
#!/bin/bash

# test-integer: evaluate the value of an integer.

INT=-5

if [ -z "$INT" ]; then
        echo "INT is empty." >&2
        exit 1
fi
if [ $INT -eq 0 ]; then
        echo "INT is zero."
else
        if [ $INT -lt 0 ]; then
                echo "INT is negative."
        else
                echo "INT is positive."
        fi
        if [ $((INT % 2)) -eq 0 ]; then
                echo "INT is even."
        else
                echo "INT is odd."
        fi
fi
```

这个脚本中最有意思的地方是，如何判断一个整数是奇数还是偶数。通过用模数 2 对数值进行求模运算，也就是将这个数值除以 2 并且返回余数，这就可以知道这个数值是奇数还是偶数了。

27.4 更现代的 test 命令版本

bash 的最近版本包括了一个符合命令，它相当于增强的 test 命令。下面是这个命令的语法。

[[*expression*]]

expression 是一个表达式，其结果为 ture 或 false。[[ ]]命令和 test 命令类似（支持所有的表达式），不过增加了一个很重要的新字符串表达式。

string1=~regex

如果 string1 与扩展的正则表达式 regex 匹配，则返回 true。这就为执行数据验证这样的任务提供了许多可能性。在前面整数表达式的例子中，如果常量 INT 含有整数以外的其他值，脚本就会执行失败。脚本需要有一种方法来验证常量包含的是否是整数。可以使用[[ ]]和=~字符串表达式操作符，按照如下方式改进脚本：

```
#!/bin/bash

# test-integer2: evaluate the value of an integer.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
        if [ $INT -eq 0 ]; then
                echo "INT is zero."
        else
                if [ $INT -lt 0 ]; then
                        echo "INT is negative."
                else
                        echo "INT is positive."
                fi
                if [ $((INT % 2)) -eq 0 ]; then
                        echo "INT is even."
                else
                        echo "INT is odd."
                fi
        fi
else
        echo "INT is not an integer." >&2
        exit 1
fi
```

通过应用正则表达式，我们可以限制 INT 的值只能是字符串，而且字符串必须以以减号（可选）开头，后面跟着一个或者多个数字。这个表达式同样消除了 INT 为空值的可能性。

[[ ]]增加的另外一个特性是==操作符支持模式匹配，就像路径名扩展那样。举例如下。

```
[me@linuxbox ~]$ FILE=foo.bar
[me@linuxbox ~]$ if [[ $FILE == foo.* ]]; then
> echo "$FILE matches pattern 'foo.*'"
> fi
foo.bar matches pattern 'foo.*'
```

这使得[[ ]] 在评估文件和路径名的时候非常有用。

27.5 (( ))——为整数设计

除了 [[ ]] 的复合命令之外，bash 同样提供了(( ))复合命令，它可用于操作整数。该命令支持一套完整的算术计算，这部分内容将在第 34 章中介绍。

(( ))用于执行算术真值测试（arithmetic truth test）。当算术计算的结果是非零值时，则算术真值测试为 true。

```
[me@linuxbox ~]$ if ((1)); then echo "It is true."; fi
It is true.
[me@linuxbox ~]$ if ((0)); then echo "It is true."; fi
[me@linuxbox ~]$
```

使用(( ))，我们可以简化 test-integer2 脚本，如下所示。

```
(#!/bin/bash

# test-integer2a: evaluate the value of an integer.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
	if ((INT == 0)); then
		echo "INT is zero."
	else
		if ((INT < 0)); then
			echo "INT is negative."
		else
			echo "INT is positive."
		fi
		if (( ((INT % 2)) == 0)); then
			echo "INT is even."
		else
			echo "INT is odd."
		fi
	fi
else
	echo "INT is not an integer." >&2
	exit 1
fi
```

注意这里使用了小于号、大于号和等号用来测试相等性。在处理整数时，这些语法看起来也更自然。另外，由于(( ))复合命令只是 shell 语法的一部分，而非普通的命令，并且只能处理整数，所以它能够通过名字来识别变量，而且不需要执行扩展操作。

27.6 组合表达式

我们也可以将表达式组合起来，来创建更复杂的计算。表达式是使用逻辑运算符组合起来的。在第 17 章，我们在学习 find 命令的时候已经介绍过。与 test 和 [[ ]]命令配套的逻辑运算符有三个，它们是 AND、OR 和 NOT。test 和[[ ]]使用不同的操作符来表示这三种逻辑操作，如表 27-4 所示。

表 27-4 逻辑操作符

Operation	test	[[]]and(())
AND	-a	&&
OR	-o	||
NOT	!	!

下面是一个 AND 运算的例子。这个脚本用来检测一个整数是否属于某个范围内的值。

```
#!/bin/bash

# test-integer3: determine if an integer is within a

# specified range of values.

MIN_VAL=1
MAX_VAL=100

INT=50

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
        if [[ INT -ge MIN_VAL && INT -le MAX_VAL ]]; then
                echo "$INT is within $MIN_VAL to $MAX_VAL."
        else
                echo "$INT is out of range."
        fi
else
        echo "INT is not an integer." >&2
        exit 1
fi
```

在这个脚本中，检测整数 INT 的值是否在 MIN_VAL 和 MAX_VAL 之间。这是通过一个[[ ]]运算符来执行的，该运算符中包含两个用“&&”运算符分隔来的表达式。当然我们也可以使用 test 完成该功能。

```
if [ $INT -ge $MIN_VAL -a $INT -le $MAX_VAL ]; then
        echo "$INT is within $MIN_VAL to $MAX_VAL."
else
        echo "$INT is out of range."
fi
```

“!”否定运算符对表达式的运算结果取反。如果表达式为 false，则返回 true；反之，如果表达式为 true，则返回 false。在下面的脚本中，我们将修改判断逻辑，以判断 INT 的值是否在指定的范围之外：

```
#!/bin/bash

# test-integer4: determine if an integer is outside a
# specified range of values.

MIN_VAL=1
MAX_VAL=100

INT=50

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
        if [[ ! (INT -ge MIN_VAL && INT -le MAX_VAL) ]]; then
                echo "$INT is outside $MIN_VAL to $MAX_VAL."
        else
                echo "$INT is in range."
        fi
else
        echo "INT is not an integer." >&2
        exit 1
fi
```

当然我们也可以使用圆括号把表达式括起来，以进行分组。如果不使用圆括号，“!”运算符仅对第一个表达式有效，而不是对两个组合后的表达式有效。用 test 命令可以按照如下进行编码：

```
if [ ! \( $INT -ge $MIN_VAL -a $INT -le $MAX_VAL \) ]; then
        echo "$INT is outside $MIN_VAL to $MAX_VAL."
else
        echo "$INT is in range."
fi
```

由于 test 使用的所有表达式和操作符都被 shell 看做命令参数（不像[[ ]]以及(( ))），因此在 bash 中有特殊含义的字符，如“<”、“>”、“(”和“)”，必须用引号括起来或者进行转义。

test 和[[ ]]命令基本上完成相同的功能，那么，哪一个更好呢？test 更为传统（而且也是 POSIX 的一部分），而[[ ]]则是 bash 特定的。由于 test 命令的使用更为广泛，因此直到如何使用 test 更为重要。但是[[ ]]命令显然更有用，而且更容易编码。

可移植性是狭隘思想的心魔

如果和“真正的”的 UNIX 人员交流，就会发现他们中的大部分并不是很喜欢 Linux。他们认为 Linux 是不纯洁、不简洁的。UNIX 信徒的一个信条是 Linux 的一切应该都是可以移植的。这意味着你编写的脚本不用修改就可以直接在任何一个类 UNIX 操作系统中运行。

UNIX 人员有足够的理由去相信这些。当在 POSIX 之前，他们看到 UNIX 命令和 shell 的专有扩展对 UNIX 世界的影响之后，自然就会担心 Linux 会对他们喜欢的操作系统产生影响。

但是可移植性有一个很严重的缺陷。它阻碍了进步，它需要使用“最小公分母”技术来完成工作。在 shell 编程中，这就意味着所有的事情要与 sh（最早的 Bourne shell）兼容。

这种缺陷是专有供应商的一个借口，它们使用该借口为自己的专有命令扩展进行辩解，也只有它们将其称之为“创新”。但它们只是为客户锁定了设备而已。

GNU 工具，比如 bash，并没有上述的限制。它通过支持标准和广泛的可用性来实现可移植性。你可以在几乎所有的操作系统中安装 bash 和其他的 GNU 工具，甚至包括 Windows，并且是免费的。所以，放心地使用 bash 的特性吧。它是真正可移植的。

27.7　控制运算符：另一种方式的分支

bash 还提供了两种可以执行分支的控制运算符。“&&”（AND）和“||”（OR）运算符与[[ ]]复合命令中的逻辑运算符类似。语法如下。

```
command1 && command2
```

和

```
command1 || command2
```

理解这两个运算符是非常重要的。对于“&&”运算符来说，先执行 command1，只有在 command1 执行成功时，command2 才能够执行。对于“||”运算符来说，先执行 command1，则只有在 command1 执行失败时，command2 才能够执行。

从实用性考虑，这意味着可以这样做：

```
[me@linuxbox ~]$ mkdir temp && cd temp
```

这会创建一个 *temp* 目录，并且当这个创建工作执行成功后，当前的工作目录才会更改为 temp。只有在第一个 mkdir 命令执行成功后，才会尝试执行第二个命令。同样，看如下命令：

```
[me@linuxbox ~]$ [ -d temp ] || mkdir temp
```

这个命令先检测 *temp* 目录是否存在，只有当检测失败时，才会创建这个目录。这种构造类型可以轻松处理脚本中的错误，后面章节会对其详细讲解。例如，我们可以在一个脚本中这样做：

```
[ -d temp ] || exit 1
```

如果脚本需要目录 *temp*，而这个目录不存在，则这个脚本会终止，并且退出状态为 1。

27.8 本章结尾语

本章以一个问题开始。我们怎样让 sys_info_page 脚本检测用户是否具有读取所有主目录的权限呢？通过我们的 if 知识，我们在 report_home_ space 函数中增加以下代码段，就可以解决这个问题。

```
report_home_space () {
	if [[ $(id -u) -eq 0 ]]; then
		cat <<- _EOF_
			<H2>Home Space Utilization (All Users)</H2>
			<PRE>$(du -sh /home/*)</PRE>
			_EOF_
	else
		cat <<- _EOF_
			<H2>Home Space Utilization ($USER)</H2>
			<PRE>$(du -sh $HOME)</PRE>
			_EOF_
	fi
	return
}
```

我们计算了 id 命令的输出。通过使用-u 选项，id 命令会输出有效用户的数字用户 ID 编号。而超级用户的编号总是 0，其他用户的编号则是一个大于 0 的数字。知道了这点以后，我们就可以创建两个不同的 here 文档，一个使用的是超级用户的权限，而另一个则受限于用户自己的主目录。

有关 sys_info_page 程序的内容暂时告一段落。不过不用担心。这些内容会再次出现。同时，当我们继续当前的工作时，还将讨论一些会用得上的主题。

第 28 章 读取键盘输入

迄今为止，本书所使用的脚本都缺少了一个大多数程序所常见的功能点——交互——也就是程序与用户互动的能力。尽管很多程序不需要与用户交互，但对一些程序来说，直接从用户获取输入会比较好。以前面章节中的脚本为例。

```
#!/bin/bash

# test-integer2: evaluate the value of an integer.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
	if [ $INT -eq 0 ]; then
		echo "INT is zero."
	else
		if [ $INT -lt 0 ]; then
			echo "INT is negative."
		else
			echo "INT is positive."
		fi
		if [ $((INT % 2)) -eq 0 ]; then
			echo "INT is even."
		else
```

```
                        echo "INT is odd."
                fi
        fi
else
        echo "INT is not an integer." >&2
        exit 1
fi
```

每当用户想要改变 INT 变量的值，就必须修改脚本。若脚本能向用户请求输入，就方便多了。本章节将讲解如何向程序添加与用户交互的功能。

28.1 read——从标准输入读取输入值

内嵌命令 read 的作用是读取一行标准输入。此命令可用于读取键盘输入值或应用重定向读取文件中的一行。read 命令的语法结构如下所示。

read [-*options*] [*variable*...]

语法中 options 为表 28-1 列出的一条或多条可用的选项，而 variable 则是一到多个用于存放输入值的变量。若没有提供任何此类变量，则由 shell 变量 REPLY 来存储数据行。

表 28-1 read 选项

选项	描述
-a array	将输入值从索引为 0 的位置开始赋给 array，本书第 35 章将介绍 array 的相关内容
-d delimiter	用字符串 delimiter 的第一个字符标志输入的结束，而不是新的一行的开始
-e	使用 Readline 处理输入。此命令使用户能使用命令行模式的相同方式编辑输入
-n num	从输入中读取 num 个字符，而不是一整行
-p prompt	使用 prompt 字符串提示用户进行输入
-r	原始模式。不能将后斜线字符翻译为转义码
-s	保密模式。不在屏幕显示输入的字符。此模式在输入密码和其他机密信息时很有用处
-t seconds	超时。在 seconds 秒后结束输入。若输入超时，read 命令返回一个非 0 的退出状态
-u fd	从文件说明符 fd 读取输入，而不是从标准输入中读取

基本上，read 命令将标准输入的字段值分别赋给指定的变量。若使用 read 命令改写之前的整数验证脚本，可能如下所示。

```
#!/bin/bash

# read-integer: evaluate the value of an integer.

echo -n "Please enter an integer -> "
read int
```

```
if [[ "$int" =~ ^-?[0-9]+$ ]]; then
	if [ $int -eq 0 ]; then
		echo "$int is zero."
	else
		if [ $int -lt 0 ]; then
			echo "$int is negative."
		else
			echo "$int is positive."
		fi
		if [ $((int % 2)) -eq 0 ]; then
			echo "$int is even."
		else
			echo "$int is odd."
		fi
	fi
else
	echo "Input value is not an integer." >&2
	exit 1
fi
```

我们使用带有-n 选项（使接下来的输出在下一行显示）的 echo 命令来输出一条提示符，然后使用 read 命令给 int 变量赋值。此脚本的运行结果如下所示。

```
[me@linuxbox ~]$ read-integer
Please enter an integer -> 5
5 is positive.
5 is odd.
```

在下述脚本中，read 命令将输入值赋给多个变量。

```
#!/bin/bash

# read-multiple: read multiple values from keyboard

echo -n "Enter one or more values > "
read var1 var2 var3 var4 var5

echo "var1 = '$var1'"
echo "var2 = '$var2'"
echo "var3 = '$var3'"
echo "var4 = '$var4'"
echo "var5 = '$var5'"
```

此脚本可以给 5 个变量赋值并输入。需要注意当输入少于或多于 5 个值的时候，read 的运作方式如下。

```
[me@linuxbox ~]$ read-multiple
Enter one or more values > a b c d e
var1 = 'a'
var2 = 'b'
var3 = 'c'
var4 = 'd'
var5 = 'e'
[me@linuxbox ~]$ read-multiple
Enter one or more values > a
```

```
var1 = 'a'
var2 = ''
var3 = ''
var4 = ''
var5 = ''
[me@linuxbox ~]$ read-multiple
Enter one or more values > a b c d e f g
var1 = 'a'
var2 = 'b'
var3 = 'c'
var4 = 'd'
var5 = 'e f g'
```

若 read 命令读取的值少于预期的数目，则多余的变量值为空，而输入值的数目超出预期的结果时，最后的变量包含了所有的多余值。

如果 read 命令之后没有变量，则会为所有的输入分配一个 shell 变量：REPLY。

```
#!/bin/bash

# read-single: read multiple values into default variable

echo -n "Enter one or more values > "
read

echo "REPLY = '$REPLY'"
```

以上脚本的运行结果如下所示。

```
[me@linuxbox ~]$ read-single
Enter one or more values > a b c d
REPLY = 'a b c d'
```

28.1.1 选项

read 命令支持的选项如前面的表 28-1 所示。

使用不同的选项，read 命令可以达成不同的有趣的效果。例如，可使用-p 选项来显示提示符：

```
#!/bin/bash

# read-single: read multiple values into default variable

read -p "Enter one or more values > "

echo "REPLY = '$REPLY'"
```

使用-t 与-s 选项，可以写出读取“秘密”输入的脚本，此脚本若一定时间内没有完成输入，会造成超时。

```
#!/bin/bash

# read-secret: input a secret passphrase
```

```
if read -t 10 -sp "Enter secret passphrase > " secret_pass; then
        echo -e "\nSecret passphrase = '$secret_pass'"
else
        echo -e "\nInput timed out" >&2
        exit 1
fi
```

上述脚本会提示用户输入秘密通行短语，系统的等待时间为 10 秒。若 10 秒内用户没有完成输入，脚本以出错状态结束运行。又因为使用了-s 选项，输入的密码并不会显示到屏幕上。

28.1.2 使用 IFS 间隔输入字段

通常 shell 会间隔提供给 read 命令的内容。这也就意味着，在输入行，由一到多个空格将多个单词分隔成为分离的单项，再由 read 命令将这些单项赋值给不同的变量。此行为是由 shell 变量 IFS（Internal Field Separator）设定的。IFS 的默认值包含了空格、制表符和换行符，每一种都可以将字符彼此分隔开。

我们可以通过改变 IFS 值来控制 read 命令输入的间隔方式。例如，文件/etc/passwd 的内容使用冒号作为字段之间的间隔符。将 IFS 的值改为单个冒号即可使用 read 命令读取/etc/passwd 文件的内容，并成功将各字段分隔为不同的变量。下面的脚本就完成了此功能。

```
#!/bin/bash

# read-ifs: read fields from a file

FILE=/etc/passwd

read -p "Enter a username > " user_name

file_info=$(grep "^$user_name:" $FILE) ①

if [ -n "$file_info" ]; then
        IFS=":" read user pw uid gid name home shell <<< "$file_info" ②
        echo "User =      '$user'"
        echo "UID =       '$uid'"
        echo "GID =       '$gid'"
        echo "Full Name = '$name'"
        echo "Home Dir. = '$home'"
        echo "Shell =     '$shell'"
else
        echo "No such user '$user_name'" >&2
        exit 1
fi
```

此脚本提示用户输入系统账户的用户名，根据用户名找到/etc/passwd 文件中相应的用户记录，并输出此记录的各个字段。此脚本包含了两行有趣的代码。

第一行位于①，它将 grep 命令的结果赋值给 file_info 变量。grep 命令使用的正则表达式保证了用户名只会与/etc/passwd 文件中的一条记录相匹配。

第二行这有趣的代码位于②，是由三部分构成的，即一条变量赋值语句一条带有变量名作参数的 read 命令和一个陌生的重定向运算符。首先让我们看一下其中的变量赋值语句。

shell 允许在命令执行之前对一到多个变量进行赋值，这些赋值操作会改变接下来所执行命令的操作环境。但是赋值的效果是暂时性的，只有在命令执行周期内有效。本例中，IFS 的值被修改为一个冒号。我们也可以通过以下方式达到此效果。

```
OLD_IFS="$IFS"
IFS=":"
read user pw uid gid name home shell <<< "$file_info"
IFS="$OLD_IFS"
```

首先我们储存了 IFS 的旧值，并将新值赋给 IFS，我们执行了 read 命令最后将 IFS 恢复原值。显然，对于做相同的事情，将变量赋值语句置于执行命令前，是更为简洁的方法。

操作符“<<<”象征一条嵌入字符串。嵌入字符串与嵌入文档类似，只不过更为简短，它包含的是一条字符串。本例将/etc/passwd 文件中读取的数据输送给 read 命令。读者或许会疑惑为什么使用这么复杂的方法而不是如下方法。

```
echo "$file_info" | IFS=":" read user pw uid gid name home shell
```

原因如下。

read 不可重定向

通常 read 命令会从标准输入中获取输入，而不能采用以下方式。

```
echo "foo" | read
```

这种方法看起来可行，实则不然。看似命令能够执行成功，但是 REPLY 变量总是空值。为什么会出现这样的现象？

这主要是由于 shell 处理管道的方式。在 bash（和其他 shell，如 sh）中，管道会创造子 shell。子 shell 复制了 shell 及 pipeline 执行命令过程中使用到的 shell 环境。前例中，read 命令就是在子 shell 中执行的。

类 UNIX 系统的子 shell 会为执行进程复制所需的 shell 环境。进程结束后，所复制的 shell 环境即被销毁，这意味着子 shell 永远不会改变父类进程的 shell

环境。read 命令给变量赋值，这些变量会成为 shell 环境的一部分。而在上述范例中，read 将变量 foo 的值赋给子 shell 环境中的 REPLY 变量，但是当命令退出时，子 shell 与其环境就会被销毁，read 命令的赋值效果也就丢失了。

嵌入字符串是解决此类问题的方法之一，在第 36 章我们将介绍另外一种办法。

28.2 验证输入

在程序具备了读取键盘输入功能的同时，也带来了新的编程上的挑战——验证输入。通常来说，程序写得好与不好的差别仅仅在于程序是否能够处理突发意外情况。而意外情况经常以错误输入的形式出现。前一章节的评估程序涉及到了少量此类操作，程序检查整数变量的值并筛选出空值与非数值的字符。对所有程序接收的输入执行此类验证是防御无效数据的重要方式，且对于多用户共享的程序尤其重要。只有那些执行特殊任务且只会被作者使用一次的程序，出于经济学原理可以省略这些安全措施。尽管如此，若程序执行的是像删除文件这样的危险任务，为以防万一还是进行数据验证会比较好。

以下是一个对不同类型输入进行验证的程序。

```
#!/bin/bash

# read-validate: validate input


invalid_input () {
        echo "Invalid input '$REPLY'" >&2
        exit 1
}

read -p "Enter a single item > "

# input is empty (invalid)
[[ -z $REPLY ]] && invalid_input

# input is multiple items (invalid)
(( $(echo $REPLY | wc -w) > 1 )) && invalid_input

# is input a valid filename?
if [[ $REPLY =~ ^[-[:alnum:]\._]+$ ]]; then
        echo "'$REPLY' is a valid filename."
        if [[ -e $REPLY ]]; then
                echo "And file '$REPLY' exists."
        else
                echo "However, file '$REPLY' does not exist."
        fi
```

```
        # is input a floating point number?
        if [[ $REPLY =~ ^-?[[:digit:]]*\.[[:digit:]]+$ ]]; then
                echo "'$REPLY' is a floating point number."
        else
                echo "'$REPLY' is not a floating point number."
        fi

        # is input an integer?
        if [[ $REPLY =~ ^-?[[:digit:]]+$ ]]; then
                echo "'$REPLY' is an integer."
        else
                echo "'$REPLY' is not an integer."
        fi
else
        echo "The string '$REPLY' is not a valid filename."
fi
```

此脚本首先提示用户进行输入，然后分析确定用户的输入。脚本使用了很多本书至今为止涵盖的概念，包括 shell 函数、[[ ]]、(( ))、控制操作符&&、if 以及适量的正则表达式。

28.3 菜单

菜单驱动是一种常见的交互方式。在菜单驱动的程序会呈现给用户一系列的选项，并请求用户进行选择。例如，可想象程序呈现的菜单如下所示。

```
Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit

Enter selection [0-3] >
```

根据写 sys_info_page 程序得到的经验，我们可以构建菜单驱动的程序来执行上述菜单中各项任务。

```
#!/bin/bash

# read-menu: a menu driven system information program

clear
echo "
Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
```

```
"

read -p "Enter selection [0-3] > "

if [[ $REPLY =~ ^[0-3]$ ]]; then
        if [[ $REPLY == 0 ]]; then
                echo "Program terminated."
                exit
        fi
        if [[ $REPLY == 1 ]]; then
                echo "Hostname: $HOSTNAME"
                uptime
                exit
        fi
        if [[ $REPLY == 2 ]]; then
                df -h
                exit
        fi
        if [[ $REPLY == 3 ]]; then
                if [[ $(id -u) -eq 0 ]]; then
                        echo "Home Space Utilization (All Users)"
                        du -sh /home/*
                else
                        echo "Home Space Utilization ($USER)"
                        du -sh $HOME
                fi
                exit
        fi
else
        echo "Invalid entry." >&2
        exit 1
fi
```

脚本在逻辑上分隔为两个部分。第一部分展示了菜单并获取用户的响应，第二部分对响应进行验证并执行相应的选项。需要注意脚本中 exit 命令的使用方法。在完成选定的功能后，exit 可防止脚本继续执行多余的代码。程序多出口通常并不是一个好现象（会导致程序逻辑难以理解），但是在上述脚本中适用。

28.4 本章结尾语

本章节迈出了程序交互的第一步，允许用户通过键盘向程序提供输入数据。使用迄今为止本书讲解过的技术，用户已经能够写出很多不同功效的程序，如专门的计算程序和易用的命令行工具前端。下一章我们将在菜单驱动程序概念的基础上对其进行改进。

28.5 附加项

因为接下来的程序会变得更加复杂，所以就需要用户认真学习并完全理解本章节程序的逻辑构建方式。作为练习，用户可使用 test 命令替代“[[ ]]”复合命令来改写本章节中的程序。提示：使用 grep 命令处理正则表达式，然后评估其退出状态。这将是个很好的练习。

第29章

流控制：WHILE 和 UNTIL 循环

前面的章节我们曾构建过一个菜单驱动的程序，用来提供不同的系统信息。此程序能有效地运作，但是却存在着重大的可用性问题——程序在执行一次选择之后即终止运行。更糟糕的是，若用户做出了无效的选择，那么程序会立刻以错误状态终止，用户没有再次尝试的机会。如果能够用某种方法重构程序，让程序能够一遍一遍地展示菜单选项，一直到用户选择退出程序，这样会好得多。

本章节将讲解叫做循环的编程概念，用于重复执行部分程序。shell 为循环提供了三种复合命令。本章节会介绍其中的两种，第 33 章中介绍第三种。

29.1 循环

日常生活充满了循环。每天上班、遛狗和切胡萝卜等都需要重复一系列的步骤。就比如用伪代码来描述切胡萝卜的过程，可能会是这样的。

1. 取案板。

2. 取刀。

3. 把胡萝卜放在案板上。

4. 拿起刀。

5. 前移胡萝卜。

6. 切胡萝卜。

7. 若整根胡萝卜都切好了，就退出程序，否则继续从第 4 步开始操作。

第 4 步到第 7 步构成了一个循环。循环中的动作会一直重复，直到条件“整根胡萝卜都切好了”为真。

29.2 while

bash 可以表达出类似的过程。若要按顺序显示 1～5 这 5 个数字，就可以构建这样的 bash 脚本。

```
#!/bin/bash

# while-count: display a series of numbers

count=1

while [ $count -le 5 ]; do
        echo $count
        count=$((count + 1))
done
echo "Finished."
```

脚本的执行结果如下所示。

```
[me@linuxbox ~]$ while-count
1
2
3
4
5
Finished.
```

while 命令的语法结构如下。

```
while commands; do commands; done
```

就如同 if 命令一样，while 会判断一系列指令的退出状态。只要退出状态为 0，它就执行循环内的命令。上述脚本中，我们创建了 count 变量并赋予 count 初始值 1。while 命令会判断 test 命令的退出状态。只要 test 命令返回的退出状态为 0，

那么循环内的指令继续执行。在每个循环周期的末尾，重复执行 test 命令。在经过 6 次循环迭代后，count 变量的值增加至 6，此时 test 命令返回的退出状态就不再是 0，循环终止。接下来程序会进一步执行循环后面的语句。

我们可以使用 while 循环改进第 28 章中的 read-menu 程序。

```
#!/bin/bash

# while-menu: a menu driven system information program

DELAY=3 # Number of seconds to display results

while [[ $REPLY != 0 ]]; do
        clear
        cat <<- _EOF_
                Please Select:

                1. Display System Information
                2. Display Disk Space
                3. Display Home Space Utilization
                0. Quit
        _EOF_
        read -p "Enter selection [0-3] > "

        if [[ $REPLY =~ ^[0-3]$ ]]; then
                if [[ $REPLY == 1 ]]; then
                        echo "Hostname: $HOSTNAME"
                        uptime
                        sleep $DELAY
                fi
                if [[ $REPLY == 2 ]]; then
                        df -h
                        sleep $DELAY
                fi
                if [[ $REPLY == 3 ]]; then
                        if [[ $(id -u) -eq 0 ]]; then
                                echo "Home Space Utilization (All Users)"
                                du -sh /home/*
                        else
                                echo "Home Space Utilization ($USER)"
                                du -sh $HOME
                        fi
                        sleep $DELAY
                fi
        else
                echo "Invalid entry."
                sleep $DELAY
        fi
done
echo "Program terminated."
```

将菜单封装到 while 循环内，程序就可以在用户每次选择后重复展示菜单项。只要 REPLY 值不为 0，重复循环，展示菜单项，给用户又一次进行选择的机会。而在每次动作结束时，系统执行 sleep 命令使程序暂停几秒，让用户能看

到选择执行的结果，随后程序清空屏幕显示并再次展示菜单项。一旦 REPLY 值为 0，也就意味着用户选择了退出项，循环终止，程序进一步执行 done 之后的代码。

29.3 跳出循环

bash 提供了两种可用于控制循环内部程序流的内建命令。其中 break 命令立即终止循环，程序从循环后的语句恢复执行。continue 命令则会导致程序跳过循环剩余的部分，直接开始下一次循环迭代。下面这个版本的 while-menu 程序实际应用了 break 和 continue。

```
#!/bin/bash

# while-menu2: a menu driven system information program

DELAY=3 # Number of seconds to display results

while true; do
	clear
	cat <<- _EOF_
		Please Select:

		1. Display System Information
		2. Display Disk Space
		3. Display Home Space Utilization
		0. Quit

	_EOF_
	read -p "Enter selection [0-3] > "

	if [[ $REPLY =~ ^[0-3]$ ]]; then
		if [[ $REPLY == 1 ]]; then
			echo "Hostname: $HOSTNAME"
			uptime
			sleep $DELAY
			continue
		fi
		if [[ $REPLY == 2 ]]; then
			df -h
			sleep $DELAY
			continue
		fi
		if [[ $REPLY == 3 ]]; then
			if [[ $(id -u) -eq 0 ]]; then
				echo "Home Space Utilization (All Users)"
				du -sh /home/*
			else
				echo "Home Space Utilization ($USER)"
				du -sh $HOME
			fi
			sleep $DELAY
```

```
                        continue
                fi
                if [[ $REPLY == 0 ]]; then
                        break
                fi
        else
                echo "Invalid entry."
                sleep $DELAY
        fi
done
echo "Program terminated."
```

这个版本的程序脚本构建了一个无限循环（无限循环永远不会自动终止），利用 true 命令向 while 提供退出状态。因为 true 退出时的状态总为 0，所以循环永远都不会停止。这是一个很常见的脚本技术。因为循环永远不能自动终止，所以需要程序员提供在适当的时刻跳出循环的方式。当用户选择为 0 时，脚本使用 break 命令来终止循环。为了使脚本执行更加高效，可以在其他脚本选项的末端使用了 continue。在确认用户做出了选择后，continue 让脚本跳过了不需要执行的代码。例如，若确认用户选择了 1，那么没有必要验证其他选项了。

29.4　until

while 命令退出状态不为 0 时终止循环，而 until 命令则刚好相反。除此之外，until 命令与 while 命令很相似。until 循环会在接收到为 0 的退出状态时终止。在 while-count 脚本中，循环会一直重复到 count 变量小于等于 5。使用 until 改写脚本也可以达到相同的效果。

```
#!/bin/bash

# until-count: display a series of numbers

count=1

until [ $count -gt 5 ]; do
        echo $count
        count=$((count + 1))
done
echo "Finished."
```

将测试表达式改写为$count –gt 5，until 就可以在合适的时刻终止循环。选择使用 while 还是 until，通常取决于哪种循环能够允许程序员写出最明了的测试表达式。

29.5 使用循环读取文件

while 和 until 可处理标准输入，这让使用 while 和 until 循环处理文件成为可能。本书前面的章节曾使用过 *distros.txt* 文件，接下来的示例显示了该文件的内容。

```
#!/bin/bash

# while-read: read lines from a file

while read distro version release; do
        printf "Distro: %s\tVersion: %s\tReleased: %s\n" \
                $distro \
                $version \
                $release
done < distros.txt
```

为将一份文件重定向到循环中，我们可在 done 语句之后添加重定向操作符。循环使用 read 命令读取重定向文件中的字段。在到达文件末端之前，退出状态为 0。在读取过文件中的每一行之后，read 命令退出，此时退出状态变为非 0，循环终止。将标准输入重定向到循环中也是可以做到的。

```
#!/bin/bash

# while-read2: read lines from a file

sort -k 1,1 -k 2n distros.txt | while read distro version release; do
        printf "Distro: %s\tVersion: %s\tReleased: %s\n" \
                $distro \
                $version \
                $release
done
```

此脚本获取 sort 命令的输出并显示文本流。但需要留意的是，因为管道是在子 shell 中进行循环操作。当循环终止时，循环内部新建的变量或者是对变量的赋值效果都会丢失。

29.6 本章结尾语

在介绍过循环，讲解过分支、子进程和队列之后，本书已经讲解了编程使用的几种主要的流控制方法。bash 还提供了很多其他的方法，但这些方法也只是对这些基本概念的改进。

第30章 故障诊断

随着脚本的复杂度越来越高，当脚本出现错误或执行情况和预期不同的时候，就需要看看是哪里出的问题。本章将讲解一些脚本中常见的错误类型以及几种用于追踪和解除错误的有用技巧。

30.1 语法错误

语法错误是一种常见的错误类型，其中就包括了 shell 语句中一些元素的拼写错误。在大多数情况下，shell 会拒绝执行含有此种类型错误的脚本。

在接下来的讨论过程中，我们将使用以下脚本来演示常见的错误类型。

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ]; then
```

```
        echo "Number is equal to 1."
else
        echo "Number is not equal to 1."
fi
```

可以看到，此脚本运行正常。

```
[me@linuxbox ~]$ trouble
Number is equal to 1.
```

30.1.1 引号缺失

现在修改上述脚本，删除第一个 echo 命令后实参后的双引号。

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ]; then
        echo "Number is equal to 1.
else
        echo "Number is not equal to 1."
fi
```

观察出现的现象如下所示。

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 10: unexpected EOF while looking for matching '"'
/home/me/bin/trouble: line 13: syntax error: unexpected end of file
```

此删除操作使脚本产生了两个错误。有趣的是，错误报告指出的行并不是之前所删除双引号所在的行，而是之后的代码。可以看到，当系统读取到所删除双引号的位置之后，bash 将继续向下寻找与前双引号对应的引号，这样的行为会一直延续到 bash 找到目标，也就是在第二个 echo 命令后的第一个引号处。然后 bash 就陷入了混乱中，即 if 命令的语法结构被破坏了，fi 语句现在处于了引号标识（但是又只有一边存在引号）的字符串中。

这种类型的错误在长脚本中很难发现，而使用带有语法结构突出显示功能的编辑器能够帮助找到这类错误。如果系统配备的是完整版的 vim，可使用以下命令启用 vim 的语法结构突出显示功能。

```
:syntax on
```

30.1.2 符号缺失冗余

另一种常见的错误是如 if 或 while 这样的复合命令结构不完整。现在就删

除 if 命令中 test 部分后的分号，看看会产生什么情况。

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ] then
        echo "Number is equal to 1."
else
        echo "Number is not equal to 1."
fi
```

结果如下所示。

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 9: syntax error near unexpected token 'else'
/home/me/bin/trouble: line 9: 'else'
```

再一次，报错信息指向的代码远在实际问题之后，这个现象的机理非常有趣。事实是，if 接收一系列的命令，并评估系列中最后一个命令为退出码。在本程序中，这个命令系列只由一条命令组成，“[,” 即 test 的同义表示。“[” 命令将其之后的部分视为一系列的参数——本例中也就是 “$number”、“=”、“1” 和 “]”。在分号被删除之后，单词 then 也就被添加到了参数系列中，这在语法中也是允许的。接下来的 echo 命令也是合法的，echo 被翻译为 if 接收的命令列表中的另一条命令，并成为 if 命令的退出码。最后遇到的就是 else，但是因为 else 是 shell 的保留单词（在 shell 中有特定的含义），并不是命令的名称，所以 else 不应该出现在这里。因此才有了如上的报错信息。

30.1.3 非预期的展开

有一些脚本错误是间歇出现的。脚本有时候会运行正常，有时候又会因为展开的结果而出错。现在将缺失的分号补回并改变 number 的值为空，即可以演示这类错误。

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=
if [ $number = 1 ]; then
        echo "Number is equal to 1."
else
        echo "Number is not equal to 1."
fi
```

运行修改过的脚本得到的输出结果如下所示。

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 7: [: =: unary operator expected
Number is not equal to 1.
```

也就是一条看起来很深奥的报错信息，外加上脚本第二条 echo 命令的输出结果。造成问题的原因是 test 命令中 number 变量的展开。当命令

```
[ $number = 1 ]
```

expansion 情形为 number 变量为空，就造成了这样的情况。

```
[ = 1 ]
```

等式无效，也就产生了错误。“=”是一个二元操作符（要求符号两边都有值），但是本例中缺少了一个值，所以 test 命令要求程序改用一元操作符（如-z）。接下来，因为 test 不成立（因为上述错误），if 命令接收到了一个非零的退出码，从而执行了第二个 echo 命令。

用户可以通过在 test 命令中使用双引号引用第一个参数方式更正这个错误。

```
[ "$number" = 1 ]
```

现在展开命令，结果是这样的。

```
[ "" = 1 ]
```

“=”的前后有了正确数量的参数。除了空字符串需要引用之外，类似包含空格的文件名这样的多字符的字符串也要前后加以引号。

30.2 逻辑错误

与语法错误不同的是，逻辑错误不会阻碍脚本的运行。因为脚本包含的逻辑问题，脚本尽管可以运行但是不能产生理想的结果。可能发生的逻辑错误有非常多种，但是以下几种逻辑错误是脚本中最常见的。

- **条件表达错误**。编写代码中，很容易写出不正确的 if、then 和 else 语句，造成错误的逻辑，例如相反的或者不完整的逻辑表达。
- **“从 1 开始”错误**。在使用计数器的循环中，可能会忽略程序需要从 0 而不是 1 开始计数才能在正确的点结束循环。这种错误会导致循环次数过多，超过了预期的终点，或循环次数过少，缺失了最后一次的迭代过程。

- **非预期的情形。** 大多数此种错误来源于程序运行过程遇到了编写程序人员没有预期到的数据或环境。非预期展开也包括在内，如一个包括了空格的文件名本应该作为单个文件名存在，却扩展成为了多个命令的实参。

30.2.1 防御编程

在编程中核实各项假设是很重要的事情，这意味我们着需要对程序的退出状态以及脚本所用命令进行仔细评估。有一个真实的例子，一位倒霉的系统管理员写了一个脚本，用于对一台重要的服务器执行维护任务，这个脚本包含了这样两行代码。

```
cd $dir_name
rm *
```

只要名为 dir_name 的目录存在，以上两行代码并没有什么本质性的错误。但是，如果目标不存在又会发生什么呢？此情形下，cd 命令跳转失败，脚本继续读取下一行代码，删除的就是当前的工作目录。完全不是管理员想要的结果！出于这样的设计，这位管理员不幸地删除了服务器一个重要的部分。

现在就看看改进此设计的一些方法。第一种可能的方法是使执行 rm 命令以 cd 的成功执行为前提。

```
cd $dir_name && rm *
```

这样以来，如果 cd 命令跳转失败，rm 命令就不会执行。情况有所改善，但是仍然存在当 dir_name 为空的情况下，用户的主目录被删除的可能性。检查 dir_name 是否确实包含有效的目录名可以避免此情形。

```
[[ -d $dir_name ]] && cd $dir_name && rm *
```

通常来说，若发生了上述情形之一，最好立刻终止脚本的运行。

```
if [[ -d $dir_name ]]; then
        if cd $dir_name; then
                rm *
        else
                echo "cannot cd to '$dir_name'" >&2
                exit 1
        fi
else
        echo "no such directory: '$dir_name'" >&2
        exit 1
fi
```

以上代码中，不仅检查 dir_name 是否包含有效的目录名，且验证了 cd 命令

是否跳转成功。任一验证没有通过的话，将对应的描述性报错信息发送给系统标准错误，且脚本终止运行，其退出状态为 1，标志着运行失败。

30.2.2 输入值验证

一条普遍适合的编程法则是若程序需获得用户输入，则程序必须能够处理任何输入值。这通常意味着必须仔细检查输入值，以保证有效的输入可用于进一步的处理过程。在 29 章讲解 read 命令时，我们曾经遇到过类似的范例。某脚本包含了以下测试来验证菜单的选择结果是否有效。

```
[[ $REPLY =~ ^[0-3]$ ]]
```

这项测试非常明确，只有用户返回的字符串是 0~3 之间的数字时，程序才会返回值为 0 的退出状态，除此之外不会接受任何其他值。有时输入值验证类型的测试写起来很有挑战性，但却是创造高质量脚本的必经之路。

时间雕琢而成的设计

当我还是一个工业设计专业的大学生时，曾听一位教授说过："工程项目设计的质量等级取决于给予设计师的时间长度。若要用五分钟来设计一个杀死苍蝇的设备，结果可能是苍蝇拍。而给予设计师五个月的话，结果就可能是激光制导的'杀苍蝇系统'"。

这项准则对编程同样适用。一方面，如果程序员只使用一次脚本，那么一个简陋劣质的脚本就能满足要求。这种类型的脚本很常见，且应开发得尽量快，使得效益最大化。而另一方面，如果脚本是用于产品，也就是说脚本会用于重要的任务，反复使用或被多个用户使用，那么脚本的开发过程就应当更加仔细小心。

30.3 测试

对任何软件的开发过程来说，包括脚本的开发过程，测试都是一个非常重要的步骤。在开源世界中有这么一种说法——"尽早发布，经常发布"。这种说法映射出了测试的重要性。通过尽早发布和经常发布新版本，软件就能够得到更多用户的使用和测试。经验表明如果能尽早在开发周期中发现 bug，那么剩余的 bug 会更加容易发现，修补 bug 的代价也会更小。

30.3.1 桩

前面讲到过使用桩（stub）核查程序流程的内容。从脚本开发的最初阶段开始，桩就是可用于查看工作进程的重要技术手段。

现在让我们回顾一下之前的文件删除问题，看看怎样提高代码的易测试性。因为初始代码的目的是删除文件，所以对代码直接进行测试具有一定的危险性，对代码进行一些修改可以解决这个问题。

```
if [[ -d $dir_name ]]; then
        if cd $dir_name; then
                echo rm * # TESTING
        else
                echo "cannot cd to '$dir_name'" >&2
                exit 1
        fi
else
        echo "no such directory: '$dir_name'" >&2
        exit 1
fi
exit # TESTING
```

因为出现错误的条件设置本身已经表达出了足够的信息，所以我们就不需要再添加什么了。最重要的变化是在 rm 命令之前放置了 echo 命令，使得 rm 命令和扩展的命令参数可以展示出来，而不是进行实际的删除操作，从而使代码能够安全地执行。在代码片段的末尾，我们添加了 exit 命令来结束测试，防止执行脚本的其他部分。是否需要添加 exit 命令取决于不同脚本的不同设计。

在测试的过程中，我们还需要添加一些扮演“记录者”角色的注释，来记录做出的改变。在测试结束后，可根据注释还原代码。

30.3.2 测试用例

开发和应用高质量的测试用例是执行测试过程中的重要部分。这要求测试人员要很小心地选择能反映边缘测试的输入数据和操作条件。在上述代码片段（算是很简单的片段）中，我们需要测试以下三种特定条件下代码的执行情况。

- dir_name 包含的是存在的目录名。
- dir_name 包含的是不存在的目录名。
- dir_name 为空。

在这三种条件下分别执行测试，就能够得到较高的测试覆盖率。

就像设计一样，测试也是由时间雕琢而成。并不是脚本的每个角落都需要测试，要紧的是确定什么是最重要的。因为出错的代码可能会造成毁灭性的后果，所以不管是代码设计还是测试都需要程序员仔细地斟酌。

30.4 调试

如果测试揭露出脚本存在问题，那么下一步就是调试。"问题"通常意味着，在某些情况下，脚本的运行结果和预期的效果不同。如果是这样的话，就需要仔细查明脚本实际上是怎样运作的以及为什么会出现这样的情况。寻找 bug 有时会需要很多侦查工作。

设计优良的脚本本身能够提供一些帮助，防御性的脚本遇到异常时会向用户反馈有用的信息。但是，解决预料之外的奇怪问题就需要介入其他的测试技术。

30.4.1 找到问题域

在一些脚本中，尤其是长脚本中，将问题相关的脚本域隔离出来是很有必要的。隔离的部分并不一定是实际问题所在之处，但是通常能提供通往实际原因的线索。其中一种能够用于隔离代码片段的方法是"注释"掉脚本的一部分。

```
if [[ -d $dir_name ]]; then
        if cd $dir_name; then
                rm *
        else
                echo "cannot cd to '$dir_name'" >&2
                exit 1
        fi
# else
#       echo "no such directory: '$dir_name'" >&2
#       exit 1
fi
```

30.4.2 追踪

bug 还经常表现为脚本中异常的逻辑流程。也就是说，脚本的一部分或者从来没有执行过，或者以错误的顺序或在错误的时间执行了。追踪是一种用于查看程序实际运行流程的技术。

一种追踪技术是通过在脚本中添加通知信息的方式来展示程序执行之处。

我们可以在上述代码片段中添加一些信息：

```
echo "preparing to delete files" >&2
if [[ -d $dir_name ]]; then
        if cd $dir_name; then
echo "deleting files" >&2
                rm *
        else
                echo "cannot cd to '$dir_name'" >&2
                exit 1
        fi
else
        echo "no such directory: '$dir_name'" >&2
        exit 1
fi
echo "file deletion complete" >&2
```

我们将这些信息发送给标准错误，从而与一般的程序输出区分开。这些信息所在的行没有缩进，使这些代码在需要删除的时候易于寻找。

在脚本执行之后，系统就已经执行了文件的删除操作。

```
[me@linuxbox ~]$ deletion-script
preparing to delete files
deleting files
file deletion complete
[me@linuxbox ~]$
```

bash 也提供了一种追踪的方法，即直接使用-x 选项或 set 命令加-x 选项。在之前的 trouble 脚本中，在脚本第一行添加-x 选项即可激活对整个脚本的追踪活动。

```
#!/bin/bash -x

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ]; then
        echo "Number is equal to 1."
else
        echo "Number is not equal to 1."
fi
```

脚本执行的结果如下所示。

```
[me@linuxbox ~]$ trouble
+ number=1
+ '[' 1 = 1 ']'
+ echo 'Number is equal to 1.'
Number is equal to 1.
```

激活追踪之后，我们就可以看到变量展开的执行情况。行开端的加号表示此

行是系统的追踪信息，以区别于一般的输出。加号是追踪信息的默认特征，是由 shell 变量 PS4（提示符字符串 4）设定的，用户可以修改变量值使追踪活动的提示符提供更多帮助信息。接下来，对 PS4 做出修改，使提示符包含执行追踪活动的脚本行号。值得注意的是，单引号使得提示符真正被使用时才展开生效。

```
[me@linuxbox ~]$ export PS4='$LINENO + '
[me@linuxbox ~]$ trouble
5 + number=1
7 + '[' 1 = 1 ']'
8 + echo 'Number is equal to 1.'
Number is equal to 1.
```

要对脚本选定的一部分而不是整个脚本执行追踪，可以使用 set 命令加-x 选项。

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

set -x # Turn on tracing
if [ $number = 1 ]; then
        echo "Number is equal to 1."
else
        echo "Number is not equal to 1."
fi
set +x # Turn off tracing
```

在这里使用 set 命令加-x 激活追踪，set 命令加+x 解除追踪。这项技术可以用来检验一个问题脚本的多个部分。

30.4.3 运行过程中变量的检验

在执行脚本并追踪脚本的过程中，显示各变量的值以体现脚本的内部活动会很有帮助。这项功能通常通过添加额外的 echo 语句完成。

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

echo "number=$number" # DEBUG
set -x # Turn on tracing
if [ $number = 1 ]; then
        echo "Number is equal to 1."
else
        echo "Number is not equal to 1."
fi
set +x # Turn off tracing
```

在这个简单的例子中，我们只输出了变量 number 的值，并使用注释标注了新行，便于之后的识别和删除。这项技术在观察脚本中的循环和计算行为时尤其有用。

30.5 本章结尾语

本章讨论了脚本开发过程中可能出现的几种问题。当然，没有涉及到的问题还有很多。本章讲述的 debug 方法已经能够帮助程序员找到大多数常见的 bug。调试是一门在实践中成长的艺术，它既包括避免 bug（在开发过程中不停测试），也包括找到 bug（有效地利用追踪技术）。

第 31 章

流控制：case 分支

本章节将继续讲解流控制的内容。第 28 章构建了一些简单的菜单，并建立了用于响应用户选择的逻辑。为达到这个目的，我们使用了一系列的 if 命令来确定哪个是选中项。这种类型的程序构造经常被用到，所以许多编程语言（包括 shell）为基于多项选择的判断提供了流控制机制。

31.1 case

bash 的多项选择复合命令被称为 case，其语法如下所示。

```
case word in
      [pattern [| pattern]...) commands ;;]...
esac
```

回顾第 28 章中的 read-menu 程序，可以看到用于响应用户选择的逻辑如下所示。

```
#!/bin/bash

# read-menu: a menu driven system information program
```

```
clear
echo "
Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"

read -p "Enter selection [0-3] > "

if [[ $REPLY =~ ^[0-3]$ ]]; then
        if [[ $REPLY == 0 ]]; then
                echo "Program terminated."
                exit
        fi
        if [[ $REPLY == 1 ]]; then
                echo "Hostname: $HOSTNAME"
                uptime
                exit
        fi
        if [[ $REPLY == 2 ]]; then
                df -h
                exit
        fi
        if [[ $REPLY == 3 ]]; then
                if [[ $(id -u) -eq 0 ]]; then
                        echo "Home Space Utilization (All Users)"
                        du -sh /home/*
                else
                        echo "Home Space Utilization ($USER)"
                        du -sh $HOME
                fi
                exit
        fi
else
        echo "Invalid entry." >&2
        exit 1
fi
```

使用 case 可以简化以上逻辑。

```
#!/bin/bash

# case-menu: a menu driven system information program

clear
echo "

Please Select:
1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"
read -p "Enter selection [0-3] > "
```

```
case $REPLY in
        0)      echo "Program terminated."
                exit
                ;;
        1)      echo "Hostname: $HOSTNAME"
                uptime
                ;;
        2)      df -h
                ;;
        3)      if [[ $(id -u) -eq 0 ]]; then
                        echo "Home Space Utilization (All Users)"
                        du -sh /home/*
                else
                        echo "Home Space Utilization ($USER)"
                        du -sh $HOME
                fi
                ;;
        *)      echo "Invalid entry" >&2
                exit 1
                ;;
esac
```

case 命令将关键字的值——本例中，即变量 REPLY 的值——与特定的模式相比较，若发现吻合的模式，就执行与此模式相联系的命令。发现吻合的模式后，将不再比对剩余的模式。

31.1.1 模式

同路径名展开一样，case 使用以“)”字符结尾的模式。表 31-1 列出一些有效的模式。

表 31-1　case 模式范例

模式	描述
a)	若关键字为 a 则吻合
[[:alpha:]])	若关键字为单个字母则吻合
???)	若关键字为三个字符则吻合
*.txt)	若关键字以.txt 结尾则吻合
*)	与任何关键字吻合。在 case 命令的最后的一个模式应用此项是个不错的做法，可对应所有前模式不吻合的关键字，也就是对应任何可能的无效值

下面是一个使用模式的范例。

```
#!/bin/bash

read -p "enter word > "

case $REPLY in
        [[:alpha:]])      echo "is a single alphabetic character." ;;
```

```
        [ABC][0-9])       echo "is A, B, or C followed by a digit." ;;
        ???)              echo "is three characters long." ;;
        *.txt)            echo "is a word ending in '.txt'" ;;
        *)                echo "is something else." ;;
esac
```

31.1.2 多个模式的组合

我们也可以使用竖线作为分隔符来组合多个模式，模式之间是“或”的条件关系。这对一些类似同时处理大写和小写字母的事件很有帮助。如下所示。

```
#!/bin/bash

# case-menu: a menu driven system information program

clear
echo "
Please Select:

A. Display System Information
B. Display Disk Space
C. Display Home Space Utilization
Q. Quit
"
read -p "Enter selection [A, B, C or Q] > "

case $REPLY in

        q|Q)    echo "Program terminated."
                exit
                ;;
        a|A)    echo "Hostname: $HOSTNAME"
                uptime
                ;;
        b|B)    df -h
                ;;
        c|C)    if [[ $(id -u) -eq 0 ]]; then
                        echo "Home Space Utilization (All Users)"
                        du -sh /home/*
                else
                        echo "Home Space Utilization ($USER)"
                        du -sh $HOME
                fi
                ;;
        *)      echo "Invalid entry" >&2
                exit 1
                ;;
esac
```

以上代码将 case-menu 程序修改为通过字母而不是数字进行菜单选择。可以注意到在新模式下，用户进行选择时可以输入大写字母，也可以输入小写字母。

31.2　本章结尾语

case 命令为用户又增添了一种新的编程技巧。在下一章中我们会看到，case 是处理特定类型问题最好的方式。

第32章 位置参数

之前我们一直没有涉及程序接收和处理命令行选项及实参的能力，本章节将讲解允许程序访问命令行内容的 shell 功能。

32.1 访问命令行

shell 提供了一组名为位置参数的变量，用于储存命令行中的关键字，这些变量分别命名为 0~9。可以通过以下方法展示这些变量。

```
#!/bin/bash

# posit-param: script to view command line parameters

echo "
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
```

```
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
"
```

这个简单的脚本展示了从变量$0 到变量$9 的值。在没有任何命令行实参的情形下执行此脚本结果如下所示。

```
[me@linuxbox ~]$ posit-param

$0 = /home/me/bin/posit-param
$1 =
$2 =
$3 =
$4 =
$5 =
$6 =
$7 =
$8 =
$9 =
```

即便没有提供任何实参，变量$0 总是会储存有命令行显示的第一项数据，也就是所执行程序所在的路径名。现在让我们，看一下提供实参情形下的程序执行结果：

```
[me@linuxbox ~]$ posit-param a b c d

$0 = /home/me/bin/posit-param
$1 = a
$2 = b
$3 = c
$4 = d
$5 =
$6 =
$7 =
$8 =
$9 =
```

注意 使用参数扩展技术，用户实际可以获取多于 9 个的参数。为标明一个大于 9 的数字，将数字用大括号括起来，例如${10}、${55}和${211}等。

32.1.1 确定实参的数目

shell 还提供了变量$#以给出命令行参数的数目。

```
#!/bin/bash

# posit-param: script to view command line parameters
echo "
Number of arguments: $#
```

```
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
"
```

以上程序运行结果如下所示。

```
[me@linuxbox ~]$ posit-param a b c d

Number of arguments: 4
$0 = /home/me/bin/posit-param
$1 = a
$2 = b
$3 = c
$4 = d
$5 =
$6 =
$7 =
$8 =
$9 =
```

32.1.2 shift——处理大量的实参

但是如果给程序提供大量的实参会发生什么呢？如下所示。

```
[me@linuxbox ~]$ posit-param *

Number of arguments: 82
$0 = /home/me/bin/posit-param
$1 = addresses.ldif
$2 = bin
$3 = bookmarks.html
$4 = debian-500-i386-netinst.iso
$5 = debian-500-i386-netinst.jigdo
$6 = debian-500-i386-netinst.template
$7 = debian-cd_info.tar.gz
$8 = Desktop
$9 = dirlist-bin.txt
```

在本示例系统中，通配符“*”扩展为 82 个实参。怎样才能处理这么多的实参呢？shell 提供了一种略显笨拙的处理方法。每次执行 shift 命令后，所有参数的值均“下移一位”。实际上，通过 shift 命令我们就可以只处理一个参数（$0 之外的一个参数，$0 值恒定）而完成全部程序任务。

```
#!/bin/bash

# posit-param2: script to display all arguments

count=1

while [[ $# -gt 0 ]]; do
        echo "Argument $count = $1"
        count=$((count + 1))
        shift
done
```

每当执行一次 shift 命令时，变量$2 的值就赋给变量$1，而$3 的值则赋给变量$2，依次类推。变量$#的值同时减 1。

在程序 posit-param2 中，我们创建了一个循环，只要还存在 1 个实参，循环就不停止。首先输出当前的实参，其次，每当循环迭代一次，就将变量 count 值加 1，代表处理过的实参数目。最后，执行 shift 命令使$1 载入下一个实参的值。以下是 posit-param2 的运行范例。

```
[me@linuxbox ~]$ posit-param2 a b c d
Argument 1 = a
Argument 2 = b
Argument 3 = c
Argument 4 = d
```

32.1.3 简单的应用程序

不使用 shift，我们也可以写出使用位置参数的有用的应用程序。下面是一个简单的文件信息程序范例。

```
#!/bin/bash

# file_info: simple file information program

PROGNAME=$(basename $0)

if [[ -e $1 ]]; then
        echo -e "\nFile Type:"
        file $1
        echo -e "\nFile Status:"
        stat $1
else
        echo "$PROGNAME: usage: $PROGNAME file" >&2
        exit 1
fi
```

这个程序输出了单个特定文件的文件类型（来自 file 命令）和文件状态（来自 stat 命令）。程序中的 PROGRAME 变量是一个有趣的角色，其值来自 basename

$0 命令的执行结果。basename 命令的作用是移除路径名的起始部分，只留下基本的文件名。在上述例子中，basename 移除了$0 参数中范例程序所在路径全名的起始部分，得到的值在构建程序信息——如程序末尾的使用信息——时很有用处。若使用 basename 构建信息，在重命名脚本后，这条信息会自动调整所包含的程序名称。

32.1.4 在 shell 函数中使用位置参数

就像位置参数可用于向 shell 脚本传递实参一样，位置参数也可用于 shell 函数实参的传递。现在将 file_info 脚本改写为 shell 函数，以进行如下演示说明。

```
file_info () {

	# file_info: function to display file information

	if [[ -e $1 ]]; then
		echo -e "\nFile Type:"
		file $1
		echo -e "\nFile Status:"
		stat $1
	else
		echo "$FUNCNAME: usage: $FUNCNAME file" >&2
		return 1
	fi
}
```

现在，如果一个包含了 file_info 函数的脚本以一个文件名为实参调用 file_info，则此实参就被传递给 file_info 函数。

在这样的条件下，我们就可以写出很多不仅普通脚本可使用，而且.bashrc 文件也适用的有用的 shell 函数。

需要注意的是，PROGRAME 变量被换成了名为 FUNCNAME 的 shell 变量。shell 自动更新 FUNCNAME 以追踪当前执行的 shell 函数。但是变量$0 包含的总是命令行第一项的路径全名（例如，程序名称），而并不像用户可能期待的那样包含了 shell 函数的名称。

32.2 处理多个位置参数

有时我们将所有的位置参数作为一个整体来处理会比较方便。例如，为目标程序写一个包装，也就是编写一个简化了目标程序运作过程的脚本或者 shell 函数的时候，包装就供应了一系列的复杂的命令行选项，并为下一层的程

序传递所需的实参。

shell 为这项功能提供了两种特殊的参数。两种参数都能够扩展为一个完整的位置参数列，但是又有着微妙的区别。表 32-1 描述了这两种参数。

表 32-1　特殊的参数*和@

参数	描述
$*	可扩展为从 1 开始的位置参数列。当包括在双引号内时，扩展为双引号引用的由全部位置参数构成的字符串，每个位置参数以 IFS shell 变量的第一个字符（默认情况下为空格）间隔开
$@	可扩展为从 1 开始的位置参数列。当包括在双引号内时，将每个位置参数扩展为双引号引用的单独单词

下面的脚本演示了这两种特殊的参数功能。

```
#!/bin/bash

# posit-params3 : script to demonstrate $* and $@

print_params () {
        echo "\$1 = $1"
        echo "\$2 = $2"
        echo "\$3 = $3"
        echo "\$4 = $4"
}
pass_params () {
        echo -e "\n" '$* :'; print_params $*
        echo -e "\n" '"$*" :'; print_params "$*"
        echo -e "\n" '$@ :'; print_params $@
        echo -e "\n" '"$@" :'; print_params "$@"
}
pass_params "word" "words with spaces"
```

在这个较为复杂的程序中，我们创建了两个实参——word 和 words with spaces，并将其传递给 pass_params 函数。而 pass_params 函数使用 4 种“$*”和“$@”的方法，将这两个实参分别依次传递给 print_params 函数。脚本执行的结果反映了它们之间的区别。

```
[me@linuxbox ~]$ posit-param3

 $* :
$1 = word
$2 = words
$3 = with
$4 = spaces

 "$*" :
$1 = word words with spaces
$2 =
$3 =
```

```
$4 =

 $@ :
$1 = word
$2 = words
$3 = with
$4 = spaces

 "$@" :
$1 = word
$2 = words with spaces
$3 =
$4 =
```

对 word 和 words with spaces 两个实参，"$*" 和 "$@" 都产生了包含 4 个单词的结果，即 word、words、with 和 spaces。"$*" 产生的是只包含一条字符串的结果，即 word words with spaces。"$@" 产生的则是包含了两条字符串的结果，即 word 和 words with spaces。

结果与实际期望相符。从中我们得到的结论就是尽管 shell 提供了四种不同的获得位置参数列的方法，因为 "$@" 保持了每个位置参数的完整性，所以迄今为止，"$@" 是大多数情况下最令人满意的方法。

32.3 更完整的应用程序

在很长的间隔之后，现在我们重新回到 sys_info_page 程序的工作上来。接下来我们将为程序添加如下所示的命令行选项。

- **输出文件**。指定程序输出文件的选项。使用形式是-f *file* 或者- -file *file*。
- **交互模式**。这个选项会提示用户输出文件的名称，并检验此文件是否已经存在。若文件已经存在，在覆盖文件之前提示用户。使用形式是-i 或者- -interactive。
- **帮助**。形式为-h 或者- -help，可使程序输出帮助性质的使用说明。

下面是完善命令行处理功能所需的代码。

```
usage () {
        echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
        return
}

# process command line options

interactive=
filename=
```

```
while [[ -n $1 ]]; do
	case $1 in
		-f | --file)		shift
					filename=$1
					;;
		-i | --interactive)	interactive=1
					;;
		-h | --help)		usage
					exit
					;;
		*)			usage >&2
					exit 1
					;;
	esac
	shift
done
```

首先，我们添加名为 usage 的 shell 函数，在使用 help 选项或者未知选项时，usage 会输出相应的提示信息。

接下来，我们将处理过程循环启动。只要位置参数$1 不为空，循环不停。在循环的末尾，使用 shift 命令推进位置参数处理进程，确保不是死循环。

在循环中，我们使用 case 语句来检验当前位置参数是否与任何程序支持的选项相匹配。若发现支持的参数选项，则基于参数做出相应的行为。若没有发现匹配项，则输出程序使用信息，脚本以 error 结束运行。

程序中-f 参数的处理过程很有趣。在检测到-f 后，即执行附加的 shift 命令，使位置参数$1 的内容置换为-f 选项后的文件名实参。

下一步我们将添加完善交互模式所需的代码。

```
# interactive mode

if [[ -n $interactive ]]; then
	while true; do
		read -p "Enter name of output file: " filename
		if [[ -e $filename ]]; then
			read -p "'$filename' exists. Overwrite? [y/n/q] > "
			case $REPLY in
				Y|y)	break
					;;
				Q|q)	echo "Program terminated."
					exit
					;;
				*)	continue
					;;
			esac
		elif [[ -z $filename ]]; then
			continue
		else
			break
```

```
                fi
        done
fi
```

若 interactive 变量不为空，则开启一个无限循环，循环中包含了文件名提示符和接下来的当前存在的文件处理代码。若设想的输出文件已经存在，则提示用户覆盖文件。另选文件或者退出程序。若用户选择覆盖当前存在的文件，则使用 break 跳出循环。需要注意的是，只有在用户选择覆盖当前存在的文件或退出程序的时候，case 语句才适用。任何其他的选择都会使循环继续运行并再次提示用户。

为了完善程序指定输出文件名的功能，我们必须首先将现存的写页面的代码改写为 shell 函数，这样做的原因马上就会揭晓。

```
write_html_page () {
        cat <<- _EOF_
        <HTML>
                <HEAD>
                        <TITLE>$TITLE</TITLE>
                </HEAD>
                <BODY>
                        <H1>$TITLE</H1>
                        <P>$TIME_STAMP</P>
                        $(report_uptime)
                        $(report_disk_space)
                        $(report_home_space)
                </BODY>
        </HTML>
        _EOF_
        return
}

# output html page

if [[ -n $filename ]]; then
        if touch $filename && [[ -f $filename ]]; then
                write_html_page > $filename
        else
                echo "$PROGNAME: Cannot write file '$filename'" >&2
                exit 1
        fi
else
        write_html_page
fi
```

处理-f 选项的逻辑出现在上述代码的底部。在逻辑中，首先检验文件是否存在，若存在，则检验文件是否确实可写。为达到这样的目的，执行 touch 命令并检验结果文件是否是常规文件。这两项检验考虑到了在输入的是无效路径（touch 会执行失败）和文件已存在的情况下，文件是否是常规文件。

可以看到，在实际生成页面的执行过程中调用了 write_html_page 函数，并将函数的输出定向到标准输出（filename 变量为空时）或重定向到指定的文件。

32.4 本章结尾语

在位置参数的帮助下，用户可写出功能性更强的脚本。简单来说，对于重复性的任务，位置参数使得用户可以写出很有帮助的 shell 函数，并放置在用户的.bashrc 文件中。

sys_info_page 程序在复杂性和先进性方面有了很大提升。下面是完整的程序，并突出显示了最近做出的改变。

```
#!/bin/bash

# sys_info_page: program to output a system information page

PROGNAME=$(basename $0)
TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

report_uptime () {
        cat <<- _EOF_
                <H2>System Uptime</H2>
                <PRE>$(uptime)</PRE>
                _EOF_
        return
}
report_disk_space () {
        cat <<- _EOF_
                <H2>Disk Space Utilization</H2>
                <PRE>$(df -h)</PRE>
                _EOF_
        return
}
report_home_space () {
        if [[ $(id -u) -eq 0 ]]; then
                cat <<- _EOF_
                        <H2>Home Space Utilization (All Users)</H2>
                        <PRE>$(du -sh /home/*)</PRE>
                        _EOF_
        else
                cat <<- _EOF_
                        <H2>Home Space Utilization ($USER)</H2>
                        <PRE>$(du -sh $HOME)</PRE>
                        _EOF_
        fi
        return
}
```

```
usage () {
        echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
        return
}
write_html_page () {
        cat <<- _EOF_
        <HTML>
                <HEAD>
                        <TITLE>$TITLE</TITLE>
                </HEAD>
                <BODY>
                        <H1>$TITLE</H1>
                        <P>$TIME_STAMP</P>
                        $(report_uptime)
                        $(report_disk_space)
                        $(report_home_space)
                </BODY>
        </HTML>
        _EOF_
        return
}

# process command line options

interactive=
filename=

while [[ -n $1 ]]; do
        case $1 in
                -f | --file)            shift
                                        filename=$1
                                        ;;
                -i | --interactive)     interactive=1
                                        ;;
                -h | --help)            usage
                                        exit
                                        ;;
                *)                      usage >&2
                                        exit 1
                                        ;;
        esac
        shift
done

# interactive mode

if [[ -n $interactive ]]; then
        while true; do
                read -p "Enter name of output file: " filename
                if [[ -e $filename ]]; then
                        read -p "'$filename' exists. Overwrite? [y/n/q] > "
                        case $REPLY in
                                Y|y)    break
                                        ;;
                                Q|q)    echo "Program terminated."
                                        exit
                                        ;;
                                *)      continue
                                        ;;
```

```
                        esac
                fi
        done
fi

# output html page

if [[ -n $filename ]]; then
        if touch $filename && [[ -f $filename ]]; then
                write_html_page > $filename
        else
                echo "$PROGNAME: Cannot write file '$filename'" >&2
                exit 1
        fi
else
        write_html_page
fi
```

现在这个脚本已经很不错了，但是还没有结束。在下一章中，我们将会对脚本做出最后的改进。

第33章 流控制：for 循环

本章是关于流控制的最后一章，我们将会再学习一个新的 shell 循环结构。因为 for 循环采用在循环期间进行序列处理的机制，所以它不同于 while 循环和 until 循环。事实证明，这在编程时是非常有用的。因此，for 循环在 bash 脚本编程中是一种十分流行的结构。

自然地，实现一个 for 循环应使用 for 命令。在新版 bash 语言中，for 命令存在两种形式。

33.1 for：传统 shell 形式

原始的 for 命令语法如下。

```
for variable [in words]; do
        commands
done
```

其中，variable 是一个在循环执行时会增值的变量名，words 是一列将按顺

序赋给变量 variable 的可选项，commands 部分是每次循环时都会执行的命令。

for 命令在命令行上是很有用的。很容易就可以获知它的工作方式。

```
[me@linuxbox ~]$ for i in A B C D; do echo $i; done
A
B
C
D
```

在本例中，for 顺序执行了一个包含 4 个字符的列表：A、B、C 和 D。它采用包含 4 个字符的列表，循环执行了 4 次。在循环体内部 echo 命令显示变量 i 的值，以此表明赋值过程。像 while 循环和 until 循环一样，for 循环以关键词 done 结束。

for 循环真正强大的功能在于创建字符列表的方式有多种。例如，可以使用花括号扩展方式，如下所示。

```
[me@linuxbox ~]$ for i in {A..D}; do echo $i; done
A
B
C
D
```

或使用路径名扩展方式，如下所示。

```
[me@linuxbox ~]$ for i in distros*.txt; do echo $i; done
distros-by-date.txt
distros-dates.txt
distros-key-names.txt
distros-key-vernums.txt
distros-names.txt
distros.txt
distros-vernums.txt
distros-versions.txt
```

再或者使用命令形式，如下所示。

```
#!/bin/bash

# longest-word : find longest string in a file

while [[ -n $1 ]]; do
        if [[ -r $1 ]]; then
                max_word=
                max_len=0
                for i in $(strings $1); do
                        len=$(echo $i | wc -c)
                        if (( len > max_len )); then
                                max_len=$len
                                max_word=$i
                        fi
                done
                echo "$1: '$max_word' ($max_len characters)"
```

```
        fi
        shift
done
```

本例的功能是在一个文件中搜索最长的字符串。当在命令行输入一个或多个文件名时，本程序调用 strings 程序（包含在 GNU binutils 软件包里）并在每个文件中产生一个可读文本“字符”。for 循环按顺序处理每个字符，判断目前的字符是不是迄今为止找到的最长字符。当循环终止，将会打印出最长的字符。最后的字符将被打印出来。

如果 for 命令中字符部分的选项被忽略，for 循环默认处理该位置参数。我们可以使用如下方法，修改 longest-word 脚本。

```
#!/bin/bash

# longest-word2 : find longest string in a file

for i; do
        if [[ -r $i ]]; then
                max_word=
                max_len=0
                for j in $(strings $i); do
                        len=$(echo $j | wc -c)
                        if (( len > max_len )); then
                                max_len=$len
                                max_word=$j
                        fi
                done
                echo "$i: '$max_word' ($max_len characters)"
        fi
done
```

如上所示，最外层循环发生了改变，使用 for 代替了 while。因为 for 命令中语句列表采用默认值，所以使用位置参数。在循环内部，前例的变量 i 已经被变换成变量 j，并且 shift 也已弃之不用。

为什么变量名是 I?

你一定已经注意到以上每一个 for 循环中都选择使用变量 i。这是为什么呢？其实，除了惯例这个理由之外，没有其他原因。for 循环使用的变量可以是任何有效变量，不过 i 是最常见的，除此之外还有 j 和 k。

这种惯例来自于 Fortran 编程语言。在 Fortran 语言中，以字母 I、J、K、L 和 M 开头的未声明的变量自动被归类为整数，而以其他字母开头的变量被归类为实数（含十进制小数的数字）。因为当需要一个临时变量时（循环通常如此），这样做会比较省力，所以这种方式促使程序员使用变量 I、J 和 K 作为循环的变量。

为此也诞生了这样的 Fortran 双关语：“GOD 是真的（实数），除非被声

明为整数。"（译者注：GOD is real, unless declared integer. GOD 双关"变量GOD"或者"上帝"，real 双关"真的"或者"实数"。）

33.2 for：C 语言形式

最近的 bash 版本已经加入了第二种 for 命令语法，它类似于 C 语言形式，并且许多其他编程语言都支持这种形式。

```
for (( expression1; expression2; expression3 )); do
      commands
done
```

其中 expression 1、expression 2 和 expression 3 为算术表达式，commands 是每次循环都要执行的命令。

就执行结果而言，这种形式等同于如下结构。

```
(( expression1 ))
while (( expression2 )); do
       commands
        (( expression3 ))
Done
```

expression1 用来初始化循环条件，expression2 用来决定循环何时结束，expression3 在每次循环末尾执行。

这里有一个典型的应用。

```
#!/bin/bash

# simple_counter : demo of C style for command

for (( i=0; i<5; i=i+1 )); do
        echo $i
done
```

执行后，它将输出如下结果。

```
[me@linuxbox ~]$ simple_counter
0
1
2
3
4
```

本例中，expression1 对变量 i 初始化赋值为 0，只要 i 的值小于 5，expression2 就允许循环继续，expression3 在每次循环重复时使 i 的值增加 1。

每当需要数值序列时，for 的 C 语言形式就能发挥作用了。接下来的两章将学习这种情况下的几个实际应用。

33.3　本章结尾语

学习了 for 命令之后，我们将最后的改进应用到 sys_info_page 脚本中。目前，report_home_ space 函数应该是这样。

```
report_home_space () {
        if [[ $(id -u) -eq 0 ]]; then
                cat <<- _EOF_
                        <H2>Home Space Utilization (All Users)</H2>
                        <PRE>$(du -sh /home/*)</PRE>
                        _EOF_
        else
                cat <<- _EOF_
                        <H2>Home Space Utilization ($USER)</H2>
                        <PRE>$(du -sh $HOME)</PRE>
                        _EOF_
        fi
        return
}
```

接下来，我们将重写上述脚本，使其可以为每个用户的主目录提供更多的细节，并且包含每个用户的文件总数和子目录总数。

```
report_home_space () {

        local format="%8s%10s%10s\n"
        local i dir_list total_files total_dirs total_size user_name

        if [[ $(id -u) -eq 0 ]]; then
                dir_list=/home/*
                user_name="All Users"
        else
                dir_list=$HOME
                user_name=$USER
        fi

        echo "<H2>Home Space Utilization ($user_name)</H2>"

        for i in $dir_list; do

                total_files=$(find $i -type f | wc -l)
                total_dirs=$(find $i -type d | wc -l)
                total_size=$(du -sh $i | cut -f 1)
                echo "<H3>$i</H3>"
                echo "<PRE>"
                printf "$format" "Dirs" "Files" "Size"
                printf "$format" "----" "-----" "----"
                printf "$format" $total_dirs $total_files $total_size
                echo "</PRE>"
```

```
        done
        return
}
```

这次重写应用了很多目前为止学习过的知识。虽然仍要求超级用户运行，但是设置了一些随后在一个 for 循环中用到的变量，而不是运行一整套作为 if 语句部分的程序。此外，在函数中我们加入了几个局部变量，并利用 printf 按格式输出内容。

第34章 字符串和数字

计算机程序其实就是处理数据。前面的章节主要从文件层面讲解了数据的处理。然而，很多编程问题需要用到更小的数据单元，例如字符串和数字，来解决。

本章将学习几个用于操纵字符串和数字的 shell 脚本特性。shell 提供了多种字符串操作的参数扩展。除了算术扩展（在第 7 章讲到），还有一个常见的名为 bc 的命令行程序，它能执行更高层次的数学运算。

34.1 参数扩展（Parameter Expansion）

虽然参数扩展在第 7 章就已出现，但是因为大部分参数扩展使用在脚本文件，而非命令行中，所以我们未加详细解释，在这之前已经使用了某些形式的参数扩展，例如 shell 变量。shell 提供了多种参数的扩展形式。

34.1.1 基本参数

参数扩展的最简单形式体现在平常对变量的使用中。举例来说，$a 扩展后成为变量 a 所包含的内容，无论 a 包含什么。简单参数也可以被括号包围，例如${a}。这对扩展本身毫无影响，但是，当变量相邻于其他文本时，则必须使用括号，否则可能让 shell 混淆。看下面这个例子，我们试图以附加字符串_file 到变量 a 内容后的方式新建一个文件名。

```
[me@linuxbox ~]$ a="foo"
[me@linuxbox ~]$ echo "$a_file"
```

由于 shell 会试图扩展名为 a_file 的变量而不是 a 变量，所以如果按序执行这些命令，结果将是一无所获。这个问题可以通过加上括号加以解决。

```
[me@linuxbox ~]$ echo "${a}_file"
foo_file
```

同样可见，大于 9 的位置参数可以通过给相应数字加上括号来访问。例如，访问第 11 个位置参数，可以这样做——${11}。

34.1.2 空变量扩展的管理

有的参数扩展用于处理不存在的变量和空变量。这些参数扩展在处理缺失的位置参数和给参数赋默认值时很有用处。这种参数扩展形式如下。

```
${parameter:-word}
```

如果 parameter 未被设定（比如不存在）或者是空参数，则其扩展为 word 的值；如果 parameter 非空，则扩展为 parameter 的值。

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:-"substitute value if unset"}
substitute value if unset
[me@linuxbox ~]$ echo $foo
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:-"substitute value if unset"}
bar
[me@linuxbox ~]$ echo $foo
Bar
```

以下是另一种扩展形式，在里面使用等号，而非连字符号。

```
${parameter:=word}
```

如果 parameter 未被设定或者为空，则其扩展为 word 的值；此外，word 的

值也将赋给 parameter。如果 parameter 非空，则扩展为 parameter 的值。

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:="default value if unset"}
default value if unset
[me@linuxbox ~]$ echo $foo
default value if unset
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:="default value if unset"}
bar
[me@linuxbox ~]$ echo $foo
Bar
```

注意 位置参数和其他特殊参数不能以这种方式赋值。

我们使用一个问号，如下所示。

```
${parameter:?word}
```

如果 parameter 未设定或为空，这样扩展会致使脚本出错而退出，并且 word 内容输出到标准错误。如果 parameter 非空，则扩展结果为 parameter 的值。

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:?"parameter is empty"}
bash: foo: parameter is empty
[me@linuxbox ~]$ echo $?
1
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:?"parameter is empty"}
bar
[me@linuxbox ~]$ echo $?
0
```

如果我们使用一个加号，如下所示：

```
${parameter:+word}
```

若 parameter 未设定或为空，将不产生任何扩展。若 parameter 非空，word 的值将取代 parameter 的值；然而，parameter 的值并不发生变化。

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:+"substitute value if set"}

[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:+"substitute value if set"}
substitute value if set
```

34.1.3 返回变量名的扩展

shell 具有返回变量名的功能。这种功能在相当特殊的情况下才会被用到。

```
${!prefix*}
${!prefix@}
```

该扩展返回当前以 prefix 开头的变量名。根据 bash 文档，这两种扩展形式执行效果一模一样。下面的例子中，我们列出了环境变量中所有以 BASH 开头的变量。

```
[me@linuxbox ~]$ echo ${!BASH*}
BASH BASH_ARGC BASH_ARGV BASH_COMMAND BASH_COMPLETION BASH_COMPLETION_DIR
BASH_LINENO BASH_SOURCE BASH_SUBSHELL BASH_VERSINFO BASH_VERSION
```

34.1.4 字符串操作

对字符串的操作，存在着大量的扩展集合。其中一些扩展尤其适用于对路径名的操作。扩展式

```
${#parameter}
```

扩展为 parameter 内包含的字符串的长度。一般来说，参数 parameter 是个字符串。然而，如果参数 parameter 是“@”或“*”，那么扩展结果就是位置参数的个数。

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo "'$foo' is ${#foo} characters long."
'This string is long.' is 20 characters long.
```

```
${parameter:offset}
${parameter:offset:length}
```

这个扩展用来提取一部分包含在参数 parameter 中的字符串。扩展以 offset 字符开始，直到字符串末尾，除非 length 特别指定。

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo ${foo:5}
string is long.
[me@linuxbox ~]$ echo ${foo:5:6}
string
```

如果 offset 的值为负，默认表示它从字符串末尾开始，而不是字符串开头。注意，负值前必须有一个空格，以防和“$”{parameter:-word}扩展混淆。如果有 length（长度）的话，length 不能小于 0。

如果参数是“@”，扩展的结果则是从 offset 开始，length 为位置参数。

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo ${foo: -5}
long.
[me@linuxbox ~]$ echo ${foo: -5:2}
lo
```

${parameter%pattern}
${parameter%%pattern}

根据 pattern 定义，这些扩展去除了包含在 parameter 中的字符串的主要部分。pattern 是一个通配符模式，类似那些用于路径名的扩展。两种形式的区别在于“#”形式去除最短匹配，而“##”形式去除最长匹配。

```
[me@linuxbox ~]$ foo=file.txt.zip
[me@linuxbox ~]$ echo ${foo#*.}
txt.zip
[me@linuxbox ~]$ echo ${foo##*.}
Zip
```

${parameter%pattern}
${parameter%%pattern}

这些扩展与上述的“#”和“##”扩展相同，除了一点——它们从参数包含的字符串末尾去除文本，而非字符串开头。

```
[me@linuxbox ~]$ foo=file.txt.zip
[me@linuxbox ~]$ echo ${foo%.*}
file.txt
[me@linuxbox ~]$ echo ${foo%%.*}
file
```

${parameter/pattern/string}
${parameter//pattern/string}
${parameter/#pattern/string}
${parameter/%pattern/string}

这个扩展在 parameter 的内容上执行搜索和替换非常有效。如果文本被发现和通配符 pattern 一致，就被替换为 string 的内容。通常形式下，只有第一个出现的 pattern 被替换。在“//”形式下，所有 pattern 都被替换。“/#”形式要求匹配出现在字符串开头，“/%”形式要求匹配出现在字符串末尾。“/string”可以省略，不过和 pattern 匹配的文本都会被删除。

```
[me@linuxbox ~]$ foo=JPG.JPG
[me@linuxbox ~]$ echo ${foo/JPG/jpg}
jpg.JPG
[me@linuxbox ~]$ echo ${foo//JPG/jpg}
jpg.jpg
[me@linuxbox ~]$ echo ${foo/#JPG/jpg}
jpg.JPG
[me@linuxbox ~]$ echo ${foo/%JPG/jpg}
JPG.jpg
```

参数扩展是一个比较重要的功能。进行字符串操作的扩展可以替代其他常用的命令，例如 set 和 cut 命令。扩展通过取代外部程序，改善了脚本的执行效率。比如，我们将改动前面章节中讨论过的 longest-word 程序，运用参数扩展${#j}代替在 subshell 中输出结果的$(echo $j | wc -c)，如下所示。

```
#!/bin/bash

# longest-word3 : find longest string in a file

for i; do
        if [[ -r $i ]]; then
                max_word=
                max_len=
                for j in $(strings $i); do
                        len=${#j}
                        if (( len > max_len )); then
                                max_len=$len
                                max_word=$j
                        fi
                done
                echo "$i: '$max_word' ($max_len characters)"
        fi
        shift
done
```

接着，我们将用 time 命令比较两个版本的效率。

```
[me@linuxbox ~]$ time longest-word2 dirlist-usr-bin.txt
dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38 characters)

real    0m3.618s
user    0m1.544s
sys     0m1.768s
[me@linuxbox ~]$ time longest-word3 dirlist-usr-bin.txt
dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38 characters)

real    0m0.060s
user    0m0.056s
sys     0m0.008s
```

原始版本的脚本花费了 3.618 秒来扫描 text 文件，而使用参数扩展的新版本只花费了 0.06 秒——这是一个比较大的改进。

34.2 算术计算和扩展

第 7 章我们学习了算术扩展，用来对整数进行算术运算。它的基本形式如下所示：

```
$((expression))
```

其中 expression 是一个有效的算术表达式。

这和用于算法计算（真实性测试）的复合命令“(( ))”有关，我们曾在第 27 章中遇到过。

通过前面章节的学习，我们已经了解了表达式和运算符的常见类型。下面，

我们将更为完整地学习它们。

34.2.1 数字进制

回顾第 9 章，我们已经学习了八进制（octal）和十六进制（hexadecimal）的数字。在算术表达式中，shell 支持任何进制表示的整数。表 34-1 列出了基本数字进制的描述。

表 34-1 不同的数字进制

符号	描述
Number	默认情况下，number 没有任何符号，将作为十进制数字
0number	在数字表达式中，以 0 开始的数字被认为是八进制数字
0xnumber	十六进制符号
base#number	base 进制的 number

看一些例子，如下所示。

```
[me@linuxbox ~]$ echo $((0xff))
255
[me@linuxbox ~]$ echo $((2#11111111))
255
```

这些例子中，我们输出了十六进制数字 ff（最大的两位数字）的值以及最大的八位数（二进制）。

34.2.2 一元运算符

有两种一元运算符：+和-。它们分别被用来指示一个数字是正或是负。

34.2.3 简单算术

表 34-2 列出了普通算术运算符。

表 34-2 算术操作符

操作符	描述
+	加法
–	减法
*	乘法
/	整除
**	求幂
%	取模（余数）

这里大部分操作符具有自描述性，但是整数除法和取模需要更深入的讨论。

由于 shell 的算术运算符仅适用于整数，除法的结果永远是完整的数字，如下所示。

```
[me@linuxbox ~]$ echo $(( 5 / 2 ))
2
```

这使得除法运算中余数的确定尤为重要，如下所示。

```
[me@linuxbox ~]$ echo $(( 5 % 2 ))
1
```

通过运用除法和取模运算，可以确定 5 被 2 整除的结果为 2，余数为 1。

计算余数在循环中很有用。它使得一个运算符能够在循环的特定间隔中执行。在下例中，我们显示了一行数字，其中 5 的倍数突出显示。

```
#!/bin/bash

# modulo : demonstrate the modulo operator

for ((i = 0; i <= 20; i = i + 1)); do
        remainder=$((i % 5))
        if (( remainder == 0 )); then
                printf "<%d> " $i
        else
                printf "%d " $i
        fi
done
printf "\n"
```

运行后，结果如下。

```
[me@linuxbox ~]$ modulo
<0> 1 2 3 4 <5> 6 7 8 9 <10> 11 12 13 14 <15> 16 17 18 19 <20>
```

34.2.4 赋值

尽管算术表达式并非立等可见，但是它们可以用来进行赋值。通过前面不同的场景，我们已经进行了多次赋值。每当赋给变量一个值时，就是赋值操作。算术表达式可以这样使用。

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo $foo

[me@linuxbox ~]$ if (( foo = 5 ));then echo "It is true."; fi
It is true.
[me@linuxbox ~]$ echo $foo
5
```

上例中，先给变量 foo 赋空值，并确认它确实为空。接着，执行一个以复

杂命令（(foo = 5)）为条件的 if 语句。整个过程有两件有趣的事。（1）它给变量 foo 赋值 5。（2）因为赋值是成功的，所以条件为 true。

注意 记住上式中“=”的确切含义很重要。单个的“=”执行赋值，即 foo = 5 意味着“让 foo 等于 5”。两个“==”用来判断是否相等，即 foo == 5 意味着“foo 是否等于 5？”这可能十分令人费解，因为 test 命令认为单个的“=”判断字符串是否相等。但这也正是另一个使用新式的“[[ ]]”和“(( ))”混合命令代替 test 的理由。

此外，除了“=”之外，shell 还提供了一些相当有用的赋值语句，如表 34-3 所示。

表 34-3 赋值操作符

运算符	描述
parameter = value	简单赋值运算。赋予 parameter 值为 value
parameter += value	加法运算。等价于 parameter = parameter+value
parameter -= value	减法运算。等价于 parameter = parameter – value
parameter *= value	乘法运算。等价于 parameter = parameter*value
parameter /= value	整除运算。等价于 parameter = parameter÷value
parameter %= value	取模运算。等价于 parameter = parameter % value
parameter++	变量后增量运算。等价于 parameter=parameter+1（查看后面的讨论）
parameter--	变量后减量运算。等价于 parameter=parameter-1
++parameter	变量前增量运算。等价于 parameter=parameter+1
--parameter	变量前减量运算。等价于 parameter=parameter-1

这些赋值操作为很多常见算术任务提供了一种快捷方式。增量（++）和减量（--）运算特别有意义，它们以 1 为间隔增加或减少参数的值。这种风格的表示法是从 C 编程语言衍生而来，并且已经被其他几种编程语言采用，其中包括 bash。

```
[me@linuxbox ~]$ foo=1
[me@linuxbox ~]$ echo $((foo++))
1
[me@linuxbox ~]$ echo $foo
2
```

这些操作既可能在参数前部也可能在参数尾部出现。虽然它们都以 1 为间隔增加或减少参数的值，两者的位置安排有一个微妙的区别。如果在参数前部，参数在返回前增加（或减少）。如果在参数尾部，该操作在参数返回后执行。这

十分奇怪，但却是故意为之。下面是一个示例。

```
[me@linuxbox ~]$ foo=1
[me@linuxbox ~]$ echo $((++foo))
2
[me@linuxbox ~]$ echo $foo
2
```

如果给变量 foo 赋值 1，然后用“++”操作增加它的值，“++”位置在参数名后，那么 foo 返回值为 1。然而，如果再次查看变量的值，会发现该值已经增加 1。如果“++”位置在参数名前，将得到比较期望的结果，如下所示。

对于大部分 shell 应用，前置运算操作则是最常用的。

“++”和“--”操作符经常和循环联合使用。下面我们将对 modulo 脚本做些改善，使它变得更为紧凑。

```
#!/bin/bash

# modulo2 : demonstrate the modulo operator

for ((i = 0; i <= 20; ++i )); do
	if (((i % 5) == 0 )); then
		printf "<%d> " $i
	else
		printf "%d " $i
	fi
done
printf "\n"
```

34.2.5 位操作

有一种操作符以一种非同寻常的方式巧妙地进行数字运算，这些操作符在位层面执行运算。它们被用于特定的低层任务中，常用来位标志的设置和读取。表 34-4 列出了位操作符。

表 34-4 位操作

操作符	描述
~	按位取反。将数字里的每一位取反
<<	逐位左移。将数字里的每一位向左移位
>>	逐位右移。将数字里的每一位向右移位
&	按位与。对数字里的每一位执行与操作
|	按位或。对数字里的每一位执行或操作
^	按位异或。对两个数字的每一位执行异或操作

注意，除了按位取反之外，也存在相应的赋值操作（例如“<<=”）。

这里将示范使用逐位左移操作，产生 2 的次方的一串数字。

```
[me@linuxbox ~]$ for ((i=0;i<8;++i)); do echo $((1<<i)); done
1
24
8
16
32
64
128
```

34.2.6 逻辑操作

正如在第 7 章中所述，“(( ))”复合命令支持多种比较操作。有另外几种可以被用于判断逻辑是否成立的操作。表 34-5 列出了完整的清单。

表 34-5　Comparison Operators

操作符	描述
<=	小于或等于
>=	大于或等于
<	小于
>	大于
==	等于
!=	不等于
&&	逻辑与
||	逻辑或
expr1?expr2:expr3	比较（三元组）操作。如果表达式 expr1 非零（算术为 true），那么执行 expr2，否则执行 expr3

当使用逻辑操作时，表达式遵循算术逻辑的规则。这就是说，值为零的表达式为 false，而非零表达式为 true。如下所示，“(( ))”复合命令将结果映射到 shell 的正常退出代码。

```
[me@linuxbox ~]$ if ((1)); then echo "true"; else echo "false"; fi
true
[me@linuxbox ~]$ if ((0)); then echo "true"; else echo "false"; fi
false
```

最奇怪的逻辑操作是三元操作。这个操作（该操作模仿 C 编程语言中的相应操作）执行一个独立的逻辑测试。它可以被用作某种意义上的 if/then/else 语

句。它作用于三个算术表达式上（不可以是字符串），并且如果第一个表达式为真（或非零），就执行第二个表达式，否则执行第三个表达式。我们可以在命令行上尝试一下。

```
[me@linuxbox ~]$ a=0
[me@linuxbox ~]$ ((a<1?++a:--a))
[me@linuxbox ~]$ echo $a
1
[me@linuxbox ~]$ ((a<1?++a:--a))
[me@linuxbox ~]$ echo $a
0
```

这是一个实际的三元操作，本例实现了一个来回切换。每次操作被执行，变量 a 的值从 0 变为 1，或从 1 变为 0。

请注意在表达式内的赋值操作并不能简单使用。当试图这样做时，bash 将输出一个错误。

```
[me@linuxbox ~]$ a=0
[me@linuxbox ~]$ ((a<1?a+=1:a-=1))
bash: ((: a<1?a+=1:a-=1: attempted assignment to non-variable (error token is
"-=1")
```

这个问题可以通过使用括号包围赋值表达式来解决。

```
[me@linuxbox ~]$ ((a<1?(a+=1):(a-=1)))
```

接下来，我们将研究一个更为综合的例子，该例在脚本中使用算术操作产生一个简单的数字表。

```
#!/bin/bash

# arith-loop: script to demonstrate arithmetic operators

finished=0
a=0
printf "a\ta**2\ta**3\n"
printf "=\t====\t====\n"

until ((finished)); do
        b=$((a**2))
        c=$((a**3))
        printf "%d\t%d\t%d\n" $a $b $c
         ((a<10?++a:(finished=1)))
done
```

在这个脚本中，基于 finished 变量的值实现了一个 until 循环。最初，该变量被设为 0（算术假），循环继续，直到变量成为非零值。在循环体内，计算计数变量 a 的平方和立方。在循环最后，判断计数变量的值。如果它小于 10（最大迭代次数），就加 1，否则给变量 finished 赋值 1，使得 finished 算术为真，从而

终止循环。运行脚本，将得到如下结果。

```
[me@linuxbox ~]$ arith-loop
a       a**2    a**3
=       ====    ====
0       0       0
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
6       36      216
7       49      343
8       64      512
9       81      729
10      100     1000
```

34.3 bc：一种任意精度计算语言

我们已经了解了 shell 可以处理所有种类的整数运算，但是如果需要执行更高级的数学运算，或者甚至使用浮点数怎么办呢？答案是无法实现，至少无法用 shell 直接实现。为了达到这个目的，我们需要使用一个外部程序。这里有几种方法可以使用，如嵌入 Perl 或 AWK 是一个可能的解决方案。但不幸的是，这已超出本书范围。

另一种方法是使用一个专门的计算器程序，大多数 Linux 系统都支持这种程序 bc。

bc 程序读取一个使用类 C 语言编写的程序文件，并执行它。bc 脚本可以是一个单独的文件，也可以从标准输入中读取。bc 语言支持很多功能，包括变量、循环以及由程序员自定义的函数。在这里我们不会完整地涵盖 bc 的知识点，而只是抛砖引玉。bc 的 man 手册已有充分详细的说明。

以一个浅显易懂的例子开始，我们下面将写一个 2 加 2 的 bc 脚本。

```
/* A very simple bc script */

2 + 2
```

脚本的第一行是一个注释。bc 使用和 C 编程语言同样的注释语法。注释可以跨越多行，以/“*”开始，以“*/”结束。

34.3.1 bc 的使用

如果将上述 bc 脚本保存为 foo.bc，那么可以这样运行它。

```
[me@linuxbox ~]$ bc foo.bc
bc 1.06.94
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation,
Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
4
```

如果仔细看看，可以在最底部版权信息的后面看到运行结果。这些信息可以通过-q (quiet)选项禁止显示。

bc 也可以交互地使用，如下所示。

```
[me@linuxbox ~]$ bc -q
2 + 2
4
quit
```

当交互地使用 bc 时，只需简单地输入运算值，计算结果就会立刻被显示出来。使用 bc 命令中的 quit 结束交互会话。

通过标准输入传递一个脚本到 bc 亦是可行的，如下所示。

```
[me@linuxbox ~]$ bc < foo.bc
4
```

既然支持标准输入，那么意味着可以使用嵌入文档、嵌入字符串和管道传递脚本。下面是一个嵌入字符串的例子。

```
[me@linuxbox ~]$ bc <<< "2+2"
4
```

34.3.2 脚本例子

作为一个实际的例子，我们将构建一个常见运算的脚本——按月偿还贷款。以下脚本使用嵌入文档传递脚本到 bc。

```
#!/bin/bash

# loan-calc : script to calculate monthly loan payments

PROGNAME=$(basename $0)

usage () {
        cat <<- EOF
        Usage: $PROGNAME PRINCIPAL INTEREST MONTHS

        Where:

        PRINCIPAL is the amount of the loan.
```

```
        INTEREST is the APR as a number (7% = 0.07).
        MONTHS is the length of the loan's term.

        EOF
}
if (($# != 3)); then
        usage
        exit 1
fi

principal=$1
interest=$2
months=$3

bc <<- EOF
        scale = 10
        i = $interest / 12
        p = $principal
        n = $months
        a = p * ((i * ((1 + i) ^ n)) / (((1 + i) ^ n) - 1))
        print a, "\n"
EOF
```

执行后，结果如下。

```
[me@linuxbox ~]$ loan-calc 135000 0.0775 180
1270.7222490000
```

本例计算了$135,000 贷款的月付款数，180 个月（15 年）的年度百分率为 7.75%。注意答案的精度，它是由赋给 bc 脚本中 scale 特定变量的值决定的。bc 的 man 手册提供了 bc 脚本编程语言的完整描述。虽然它的数学表示法与 shell 的稍微不同（bc 更接近于 C 语言），但是就本书目前涵盖的知识而言，大部分内容都是相似的。

34.4 本间结尾语

本章我们学习了脚本中许多“实用的”小技巧。随着脚本编程经验的增长，有效而巧妙地操纵字符串和数字将会是极其珍贵的。本书中的 loan-calc 脚本，说明了哪怕是最简单的脚本也可以做一些真正有用的事。

34.5 附加项

虽然 loan-calc 脚本的基本功能已经实现，但是这个脚本远非完善。读者可以试着改善 loan-calc 脚本，使其包括下面的功能。

- 对命令行参数的充分验证。
- 用来实现“交互”模式的命令行选项，该模式提示用户输入贷款的本金、利率和偿还期。
- 更好的输出格式。

第35章 数组

在上一章中，我们了解了 shell 如何操作字符串和数字。到目前为止，我们所接触到的数据类型在计算机科学领域被称为标量变量（scalar variable），也就是说，该变量包含一个单一值。

在本章中，我们将会学习一种包含多个值的数据结构——数组。事实上，数组几乎是所有程序设计语言的一大特点。尽管 shell 对数组的支持有限，但它对于解决程序设计问题是非常有帮助的。

35.1 什么是数组

数组是可以一次存放多个值的变量，数组的组织形式如同表格一样。下面以电子表格为例。一个电子表格就像一个二维数组一样。它由行和列组成，根据行与列的地址可以在电子表格里标识每一个独立单元的位置。数组也是以这种方式工作的。数组中的单元叫做元素，并且每个元素中含有数据。使用一种

叫做索引或是下标的地址就可以访问一个独立的数组元素。

大多数的程序设计语言支持多维数组。电子表格就是多维数组的一个实例，该数组是由宽度和高度两个维度组成的二维数组。尽管最经常使用的可能是二维和三维数组，但是很多语言支持任意维数的数组。

bash 中的数组是一维的。可以将它想象成只有一列的电子表格。尽管有这个限制，但是它们还是有很多的应用。bash 的第二个版本开始提供对数据的支持。而最初的 UNIX shell 程序 sh 是不支持数组的。

35.2 创建一个数组

命名数组变量同命名其他 bash 变量一样，当访问数组变量时可以自动创建它们。实例如下。

```
[me@linuxbox ~]$ a[1]=foo
[me@linuxbox ~]$ echo ${a[1]}
foo
```

这里我们看到的是赋值和访问数组元素的例子。通过第一条命令，可将值 foo 赋给数组 a 的元素 1。第二条命令显示了元素 1 的存储值。在第二条命令中使用花括号是为了阻止 shell 在数组元素名里试图扩展路径名。

使用 declare 命令也可以创建数组，如下所示。

```
[me@linuxbox ~]$ declare -a a
```

这是使用选项-a 和 declare 创建数组 a 的实例。

35.3 数组赋值

赋值的方式可以有两种。使用下面的语法可以赋单一值。

```
name[subscript]=value
```

这里的 name 是数组名，并且 subscript 是大于或等于 0 的整数（或算术表达式）。要注意的是，数组的第一个元素是 subscript0，而不是 1。value 是赋给数组元素的字符串或整数。

使用下面的语法可以赋多值。

```
name=(value1 value2...)
```

这里的 name 是数组名，并且将 value1 value2...等值依次赋予从元素 0 开始的数组元素。例如，如果想要将一星期中天数的缩写赋给数组 days，那么我们可以像下面这样赋值。

```
[me@linuxbox ~]$ days=(Sun Mon Tue Wed Thu Fri Sat)
```

通过为每个值指定一个下标来给特定元素赋值也是可行的。

```
[me@linuxbox ~]$ days=([0]=Sun [1]=Mon [2]=Tue [3]=Wed [4]=Thu [5]=Fri [6]=Sat)
```

35.4 访问数组元素

那么数组有哪些应用呢？就像使用电子表格程序可以执行很多数据管理任务一样，使用数组可以完成很多程序设计任务。

以一个简单的数据采集和表示为例。创建一个脚本，用于校验特定目录中文件的修改次数。从这些数据来看，脚本将会输出一个表格来显示文件最后一次修改发生在一天中的什么时候。使用这样的一个脚本可以来检测什么时候系统是最活跃的。这个称之为 hours 的脚本产生的结果如下。

```
[me@linuxbox ~]$ hours .
Hour        Files     Hour      Files
----        -----     ----      -----
00          0         12        11
01          1         13        7
02          0         14        1
03          0         15        7
04          1         16        6
05          1         17        5
06          6         18        4
07          3         19        4
08          1         20        1
09          14        21        0
10          2         22        0
11          5         23        0
Total files = 80
```

执行 hours 程序，并指定当前目录为目标目录，将产生一个表格用来显示在一天中每个小时（0～23）有多少文件经过最后一次修改。产生表格的代码如下。

```
#!/bin/bash

# hours : script to count files by modification time

usage () {
        echo "usage: $(basename $0) directory" >&2
}
```

```
# Check that argument is a directory
if [[ ! -d $1 ]]; then
        usage
        exit 1
fi

# Initialize array
for i in {0..23}; do hours[i]=0; done

# Collect data
for i in $(stat -c %y "$1"/* | cut -c 12-13); do
        j=${i/#0}
        ((++hours[j]))
        ((++count))
done

# Display data
echo -e "Hour\tFiles\tHour\tFiles"
echo -e "----\t-----\t----\t-----"
for i in {0..11}; do
        j=$((i + 12))
        printf "%02d\t%d\t%02d\t%d\n" $i ${hours[i]} $j ${hours[j]}
done
printf "\nTotal files = %d\n" $count
```

这个脚本包含了一个函数（usage）和由 4 部分组成的一个主函数。在第一部分，我们检测到一个命令行参数并且这是一个目录。如果不是的话，显示 usage 信息，同时退出。

第二部分初始化数组 hours。通过给每个元素赋 0 值来实现。尽管在调用数组之前对数组中的元素没有特殊要求，但是脚本需要保证没有元素是空的。注意一下有趣的循环创建方式，通过使用花括号扩展（{0...23}），能够很容易地为 for 循环生成一系列初始值。

第三部分通过运行 stat 程序遍历目录中每个文件来采集数据。使用 cut 选项从结果中提取两位数的小时数（hour）。在循环内部，需要清除 hour 域中的前导值 0，这是因为 shell 将要试图（最终失败了）以八进制的形式来表示数值 00~09（见表 34-1）。接下来，将与一天中的小时数相对应的数组元素值增加 1。最后，使用一个计数器（count）来追踪目录里文件的总数目。

脚本的最后一部分显示了数组的内容。首先我们输出几个标题行，然后，进入一个产生两列输出的循环。最后，输出文件的最终统计结果。

35.5 数组操作

有很多常见的数组操作。比如删除数组、确定数组大小和排序等在脚本中

有很多应用。

35.5.1 输出数组的所有内容

我们可以使用下标“*”和“@”来访问数组中的每个元素。对于定位参数来讲，符号“@”较之更有用。例证如下。

```
[me@linuxbox ~]$ animals=("a dog" "a cat" "a fish")
[me@linuxbox ~]$ for i in ${animals[*]}; do echo $i; done
a
dog
a
cat
a
fish
[me@linuxbox ~]$ for i in ${animals[@]}; do echo $i; done
a
dog
a
cat
a
fish
[me@linuxbox ~]$ for i in "${animals[*]}"; do echo $i; done
a dog a cat a fish
[me@linuxbox ~]$ for i in "${animals[@]}"; do echo $i; done
a dog
a cat
a fish
```

我们创建了数组 animals，并使用 3 个双单词字符串为其赋值，然后执行 4 个循环以便观察单词拆分对数组内容的影响。如果对符号${animals[*]}和${animals[@]}加以引用，就会得到不同的结果。符号“*”将数组所有内容放在一个字中，而符号“@”使用 3 个字来显示数组的真实内容。

35.5.2 确定数组元素的数目

使用参数扩展，我们可以采用类似获取字符串长度的方式来确定数组中元素的数目。实例如下。

```
[me@linuxbox ~]$ a[100]=foo
[me@linuxbox ~]$ echo ${#a[@]} # number of array elements
1
[me@linuxbox ~]$ echo ${#a[100]} # length of element 100
3
```

我们首先创建了数组 a，并且将字符串 foo 赋给第 100 个元素。接下来，我们使用符号“@”通过参数扩展来确定数组长度。最后，我们查看包含字符串

foo 的元素 100 的长度。值得一提的是，当将字符串赋给元素 100 时，bash 报告数组中只有一个元素。这与其他一些编程语言的行为是不同的。在这些语言中，数组中未使用的元素（元素 0~99）初始化为空值并参与计数。

35.5.3 查找数组中使用的下标

由于 bash 允许在下标赋值中包含“空格”，有时这对确定实际存在的元素是很有用的。这可以过参数扩展来实现，其形式如下。

```
${!array[*]}
${!array[@]}
```

这里的 array 是数组变量名。就像符号“*”和“@”等参数扩展一样，引用中含有的“@”形式是最有用的，因为它将数组内容扩展成独立的单词。

```
[me@linuxbox ~]$ foo=([2]=a [4]=b [6]=c)
[me@linuxbox ~]$ for i in "${foo[@]}"; do echo $i; done
a
bc
[me@linuxbox ~]$ for i in "${!foo[@]}"; do echo $i; done
24
6
```

35.5.4 在数组的结尾增加元素

如果在数组的结尾需要添加元素的话，知道数组中元素的数目是没有用的，因为符号“*”和“@”返回的值并不会告诉我们使用的最大数组索引是什么。幸运的是，shell 提供了一种解决方法。通过使用“+=”赋值运算符，可以在数组的尾部自动地添加元素。这里，我们将 3 个值赋给数组 foo，然后再添加 3 个元素。

```
[me@linuxbox ~]$ foo=(a b c)
[me@linuxbox ~]$ echo ${foo[@]}
a b c
[me@linuxbox ~]$ foo+=(d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
```

35.5.5 数组排序操作

就像电子表格一样，通常需要将数据列中的值进行排序。shell 虽然没有直接的方式来完成排序功能，但是用一些代码来完成并非难事。

```
#!/bin/bash

# array-sort : Sort an array
```

```
a=(f e d c b a)
echo "Original array: ${a[@]}"
a_sorted=($(for i in "${a[@]}"; do echo $i; done | sort))
echo "Sorted array: ${a_sorted[@]}"
```

当执行时，脚本会产生如下的内容。

```
[me@linuxbox ~]$ array-sort
Original array: f e d c b a
Sorted array:   a b c d e f
```

脚本巧妙地使用一个替换命令将原数组（a）的内容复制到数组（a_sorted）中。通过改变设计流程，我们可以这个基本的技术就可被用来执行数组中的多种操作。

35.5.6 数组的删除

使用 unset 命令，我们可以删除数组，如下所示。

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox ~]$ unset foo
[me@linuxbox ~]$ echo ${foo[@]}

[me@linuxbox ~]$
```

我们也可以使用 unset 来删除单个数组元素。

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox ~]$ unset 'foo[2]'
[me@linuxbox ~]$ echo ${foo[@]}
a b d e f
```

在这个例子里，我们删除了数组的第 3 个元素（下标为 2）。记住，数组是以下标 0 开始的，而不是 1 开始的！同时需要注意的是，我们必须引用数组元素来阻止 shell 执行路径名扩展。

很有趣的是，对数组元素赋一个空值并不意味着清空它的内容，如下所示。

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo[@]}
b c d e f
```

任何涉及到不含下标的数组变量的引用指的是数组中的元素 0，如下所示。

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
```

```
[me@linuxbox ~]$ foo=A
[me@linuxbox ~]$ echo ${foo[@]}
A b c d e f
```

35.6 本章结尾语

如果在 bash 手册文档中搜索 array，我们会发现在很多使用数组变量的实例。大多数实例是相当令人费解的，但是在一些特殊情况下可能会提供临时效用。事实上，在 shell 程序设计中，未能非常充分地涵盖数组的所有内容，很大程度上是因为传统的 UNIX shell 程序（比如 sh）缺乏对数组的支持。传统 shell 对数组普遍缺乏支持是很不幸的，因为数组广泛运用于其他的程序设计语言，并且提供强有力的工具来解决多种程序设计问题。

数组和循环本身具有密切的关系，并且经常被一块使用。下面的循环形式非常适合估算数组下标。

```
for ((expr1; expr2; expr3))
```

第36章 其他命令

在本书最后一章，我们将会讲解一些琐碎零散的知识。虽然我们在前面的章节中已经学习了很多知识，但是还有大量的 bash 特性没有涉及到。对于那些整合在 Linux 发行版中的 bash，其中的绝大多数是相当令人费解的，但是它们却很有帮助。除此之外，还有一些命令，虽然不经常用，却对特定的程序设计问题大有帮助。下面我们将学习这方面内容。

36.1 组命令和子 shell

bash 允许将命令组合到一起使用，这有两种方式，一种是利用组命令，另一种是使用子 shell。下面是这两种方式的语法实例。

组命令：

```
{ command1; command2; [command3; ...] }
```

子 shell：

(*command1*; *command2*; [*command3*; ...])

这两种形式的区别在于，组命令使用花括号将其命令括起来，而子 shell 则用圆括号。值得注意的是，在 bash 实现组命令时，必须使用一个空格将花括号与命令分开，并且在闭合花括号前使用分号或是换行来结束最后的命令。

36.1.1 执行重定向

那么组命令和子 shell 有什么用途呢？尽管它们有一处主要的区别（马上将会涉及这一点），但是它们都可以用来管理重定向。下面让我们看一个在多个命令中执行重定向的脚本段。

```
ls -l > output.txt
echo "Listing of foo.txt" >> output.txt
cat foo.txt >> output.txt
```

显而易见，3 条命令将输出重定向为 *output.txt* 文件。使用组命令，可以按照如下方式编码。

```
{ ls -l; echo "Listing of foo.txt"; cat foo.txt; } > output.txt
```

使用子 shell 时也是一样，如下所示。

```
(ls -l; echo "Listing of foo.txt"; cat foo.txt) > output.txt
```

使用这个技术，可以减少一些输入，但是组命令或是子 shell 真正有价值的地方在于管道的使用。当创建命令管道时，通常将多条命令的结果输出到一条流中，这很有用。组命令和子 shell 使得这一点变得简单，如下所示。

```
{ ls -l; echo "Listing of foo.txt"; cat foo.txt; } | lpr
```

这里我们已经将 3 个命令的输出进行合并，并通过管道输出到 lpr 的输入以产生一个打印报告。

36.1.2 进程替换

虽然组命令和子 shell 看起来相似，都可以用来为重定向整合流，但是，它们有一处主要的不同。子 shell（正如名字所示）在当前 shell 的子拷贝中执行命令，而组命令在当前 shell 里执行所有命令。这意味着子 shell 复制当前的环境变量以创建一个新的 shell 实例。当子 shell 退出时，复制的环境变量也就消失了，因此，任何对子 shell 环境（包括变量赋值）的改变也同样丢失了。所以，大多数情况下，除非脚本需要子 shell，否则组命令比子 shell 更可取。组命令更快，并

且需要更少的内存。

在第 28 章中，我们已经接触到了子 shell 环境存在问题的一个实例，管道中的 read 命令没有像先前预期的那样工作。我们可以采用如下的方式重新创建管道。

```
echo "foo" | read
echo $REPLY
```

REPLY 变量的内容总是为空，因为 read 命令是在子 shell 中执行的，并且当子 shell 终止的时候，REPLY 的拷贝也遭到了破坏。

由于总是在子 shell 中执行管道中的命令，任何变量赋值的命令都会遇到这个问题。很幸运的是，shell 提供了一种叫做进程替换的外部扩展方式来解决这个问题。

实现进程替换有两种方式，一种是产生标准输出的进程，如下所示。

```
<(list)
```

另一种是吸纳标准输入的进程，如下所示。

```
>(list)
```

这里的 list 是一系列的命令。

为了解决上述 read 命令的问题，我们可以像这样使用进程替换。

```
read < <(echo "foo")
echo $REPLY
```

进程替换允许将子 shell 的输出当做一个普通的文件，目的是为了重定向。事实上，这是一种扩展形式，我们可以查看它的真实值。

```
[me@linuxbox ~]$ echo <(echo "foo")
/dev/fd/63
```

通过使用 echo 来查看扩展结果，可以看到文件/dev/fd/63 正为子 shell 提供输出。

进程替换通常结合带有 read 的循环使用。这里有一个读循环的实例，该读循环用来处理子 shell 创建的目录列表的内容。

```
#!/bin/bash

# pro-sub : demo of process substitution

while read attr links owner group size date time filename; do
    cat <<- EOF
        Filename:   $filename
        Size:       $size
```

```
            Owner:      $owner
            Group:      $group
            Modified:   $date $time
            Links:      $links
            Attributes: $attr
    EOF
done < <(ls -l | tail -n +2)
```

循环在目录列表的每一行执行 read 操作。脚本的最后一行产生了列表本身。这一行将进程替换的输出重新定向到循环的标准输入。进程替换管道中的 tail 命令用来清除列表的第一行，之后这一行不再需要了。

当执行这个命令的时候，脚本产生如下的输出。

```
[me@linuxbox ~]$ pro_sub | head -n 20
Filename:    addresses.ldif
Size:        14540
Owner:       me
Group:       me
Modified:    2012-04-02 11:12
Links:       1
Attributes:  -rw-r--r-

Filename:    bin
Size:        4096
Owner:       me
Group:       me
Modified:    2012-07-10 07:31
Links:       2
Attributes:  drwxr-xr-x

Filename:    bookmarks.html
Size:        394213
Owner:       me
Group:       me
```

36.2 trap

在第 10 章中，我们了解了程序如何响应信号。同样我们也可以将这种功能应用到脚本中。虽然到目前为止，我们所写的脚本还不需要这种功能（因为它们有着更短的执行时间，并且不产生临时文件），但是，拥有一个信号处理程序可能对庞大复杂的脚本大有裨益。

当设计一个庞大复杂的脚本时，我们一定要考虑到，如果在脚本正在运行时，用户注销或是关闭电脑时会发生什么情况。当这样的情况发生时，将会把信号发送到所有受影响的进程。相应地，执行那些进程的程序能够通过一些操作来保证程序合理有序地结束。设想一下，比如说，脚本在执行的时候创建了一个临时文件。在一个好的设计中，当脚本结束工作时，该临时文件会自动删

除。如果接收的信号表明将要过早地结束程序，这个时候让脚本删除这个文件也是很明智的。

为了实现这个目的，bash 提供了一种 trap 机制。内置命令 trap 可以恰如其分地实现 trap 机制。trap 命令使用的语法如下。

```
trap argument signal [signal…]
```

这里的 argument 是作为命令被读取的字符串，而 signal 是对信号量的说明，该信号量将会触发解释命令的执行。

下面是一个简单的例子。

```
#!/bin/bash

# trap-demo : simple signal handling demo

trap "echo 'I am ignoring you.'" SIGINT SIGTERM

for i in {1..5}; do
        echo "Iteration $i of 5"
        sleep 5
done
```

每当运行中的脚本接收到 SIGINT 或者 SIGTERM 信号时，脚本定义的 trap 将执行 echo 命令。当用户通过按下 Ctrl-C 键来试图结束脚本时，程序的执行情况如下。

```
[me@linuxbox ~]$ trap-demo
Iteration 1 of 5
Iteration 2 of 5
I am ignoring you.
Iteration 3 of 5
I am ignoring you.
Iteration 4 of 5
Iteration 5 of 5
```

可以看出，每次用户试图中断程序时，都会输出这样的信息。

构造一个字符串来形成一系列有用的命令看起来是很笨拙的，因此，通常的做法是指定 shell 函数来代替命令。在下面的例子中，我们为每个将要处理的信号指定一个独立的 shell 函数。

```
#!/bin/bash

# trap-demo2 : simple signal handling demo

exit_on_signal_SIGINT () {
        echo "Script interrupted." 2>&1
        exit 0
```

```
}
exit_on_signal_SIGTERM () {
        echo "Script terminated." 2>&1
        exit 0
}

trap exit_on_signal_SIGINT SIGINT
trap exit_on_signal_SIGTERM SIGTERM

for i in {1..5}; do
        echo "Iteration $i of 5"
        sleep 5
done
```

这个脚本为两个不同的信号定义了相应的 trap 命令。当接收到特定信号的时候，每个 trap 相应地执行指定的 shell 函数。注意到每个信号处理函数中的 exit 命令。如果没有 exit 命令，脚本会循环执行该函数。

在脚本执行的过程中，当用户按下 Ctrl-C 键时，产生的结果如下。

```
[me@linuxbox ~]$ trap-demo2
Iteration 1 of 5
Iteration 2 of 5
Script interrupted.
```

临时文件

在脚本中使用信号控制句柄，是为了删除脚本执行过程中用于保存中间变量的临时文件，一些术语称其为临时文件。传统意义上来说，类 UNIX 系统中的程序在/tmp 目录里创建临时文件，该目录是用于保存临时文件的共享目录。但是，由于目录是共享的，不可避免地产生安全上的考虑，尤其是超级用户特权下运行的程序。除了为所有系统用户都可以访问的文件设置适当的权限之外，赋给临时文件一个不可预知的文件名是非常重要的。这就避免了临时快速攻击（temp race attack）的漏洞。创建一个不可预知（但是仍然是可以描述的）的名称的方法如下所示。

```
tempfile=/tmp/$(basename $o).$$.$RANDOM
```

这将会创建一个包含程序名的文件名，紧随其后的是进程 ID（PID）以及一个随机整数。但是，要注意的是，$RANDOM shell 变量返回一个范围仅在 1~32767 之间的值，在计算机领域里，这并不是一个很大的范围，因此，单一的变量实例不足以抵御一个不达目的不罢休的攻击者。

比较好的一个方法是使用 mktemp 程序（不要与 mktemp 标准库函数相混淆）来命名和创建临时文件。mktemp 程序使用模板作为参数来创建文件名。这个模板应该包括一系列的 X 字符，可以用相应的随机字母和数字来代替这

些 X 字符。X 字符的序列越长，随机字符的序列就会越长。下面是一个实例。

```
tempfile=$(mktemp /tmp/foobar.$$.XXXXXXXXXX)
```

这就创建了一个临时文件，并且将其名称赋给了变量 tempfile。随机的字母和数字代替了模板中的字符 X。那么，最终的文件名（在这个例子里，最后的文件名也包含了特殊参数$$的扩展值以获得 PID）可能会如下所示。

```
/tmp/foobar.6593.UOZuvM6654
```

尽管 mktemp 手册页声明，mktemp 构造了一个临时文件名，但是它同时也创建了这个文件。

如果是普通用户执行的脚本，更为明智的做法是避免使用/tmp 目录，而在用户主目录下而为临时文件创建一个目录，其代码如下。

```
[[ -d $HOME/tmp ]] || mkdir $HOME/tmp
```

36.3 异步执行

有的时候我们希望同时执行多项任务。众所周知，所有现代的操作系统即便不是多用户系统，至少也是多任务系统。脚本可以在多任务中运行。

这涉及到父脚本以及一个或多个子脚本的加载问题，子脚本可以在父脚本运行时执行其他额外的任务。但是，当一系列脚本以这种方式运行的时候，保持父脚本与子脚本的协调一致就会是一个问题。也就是说，试想这样一种情况，如果父脚本与子脚本彼此相互依赖，一个脚本必须等待另一个脚本任务完成之后才能继续完成自己的任务。

bash 提供了一个内置的命令来帮助管理异步执行。wait 命令可以让父脚本暂停，直到指定的进程（比如子脚本）结束。

36.3.1 wait 命令

首先，我们来演示 wait 命令。为此我们需要两个脚本来完成这个过程。下面是一个父脚本。

```
#!/bin/bash

# async-parent : Asynchronous execution demo (parent)

echo "Parent: starting..."
```

```
echo "Parent: launching child script..."
async-child &
pid=$!
echo "Parent: child (PID= $pid) launched."

echo "Parent: continuing..."
sleep 2

echo "Parent: pausing to wait for child to finish..."
wait $pid

echo "Parent: child is finished. Continuing..."
echo "Parent: parent is done. Exiting."
```

下面是一个子脚本。

```
#!/bin/bash

# async-child : Asynchronous execution demo (child)

echo "Child: child is running..."
sleep 5
echo "Child: child is done. Exiting."
```

在这个例子中，我们可以看出子脚本内容非常简单，父脚本执行实际的操作。在父脚本中，子脚本加载并在后台运行。通过将$! shell 程序参数值赋给 pid 变量来记录子脚本的进程 ID，该参数值总是包含后台中最后一次运行的进程 ID。

父脚本继续运行，随后执行带有子进程 PID 的 wait 命令。这会导致父脚本暂停，直到子脚本退出；子脚本退出之后，父脚本也结束。

当执行的时候，父脚本和子脚本产生如下的输出。

```
[me@linuxbox ~]$ async-parent
Parent: starting...
Parent: launching child script...
Parent: child (PID= 6741) launched.
Parent: continuing...
Child:  child is running...
Parent: pausing to wait for child to finish...
Child:  child is done. Exiting.
Parent: child is finished. Continuing...
Parent: parent is done. Exiting.
```

36.4 命名管道

大多数类 UNIX 系统支持创建一个叫做命名管道的特殊类型的文件。使用命名管道可以建立两个进程之间的通信，并且可以像其他类型的文件一样使用。虽然它们没有其他类型的文件那么受欢迎，但是它们仍然值得了解。

客户端/服务器模式是一种常见的程序设计结构。它可以使用命名管道这样的通信方式，也可以使用网络连接这样的进程间通信方式。

使用客户端/服务器程序设计架构最为广泛的地方当然是 Web 服务器与 Web 浏览器之间的通信。Web 浏览器充当客户机，客户机对服务器提出请求，服务器通过网页的形式对浏览器做出回应。

命名管道的工作方式与文件雷同，但实际上是两块先进先出（FIFO）的缓冲区。与普通的（未命名的）管道一样，数据从一端进入，从另一端出来。使用命名管道，也可以以如下方式设置。

```
process1 > named_pipe
```

以及

```
process2 < named_pipe
```

执行效果等效于如下语句。

```
process1 | process2
```

36.4.1 设置命名管道

首先，我们必须创建一个命名管道。使用 mkfifo 命令即可完成，如下所示。

```
[me@linuxbox ~]$ mkfifo pipe1
[me@linuxbox ~]$ ls -l pipe1
prw-r--r-- 1 me   me    0 2012-07-17 06:41 pipe1
```

这里使用 mkfifo 命令创建一个名为 pipe1 的命名管道。使用 ls 命令查看文件属性，可以看到属性字段的第一个字母是 p，这表明它是一个命名管道。

36.4.2 使用命名管道

为了说明命名管道是如何工作的，我们需要两个终端窗口（或者两个虚拟的控制台）。在第一个终端中，我们输入一条简单的命令，并将其输出重新定位到这个命名管道，如下所示。

```
[me@linuxbox ~]$ ls -l > pipe1
```

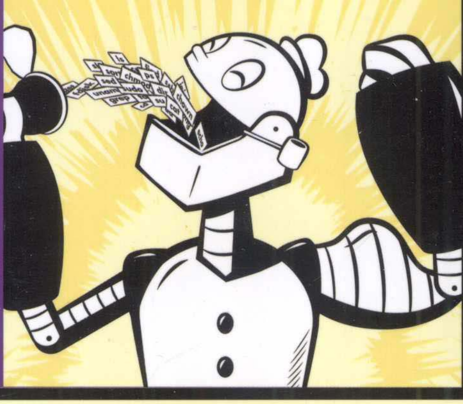
按下 Enter 键之后，这个命令看起来像挂起来了。这是因为从管道的另一端还没有接收到数据。当这种情况发生时，也就是说命名管道被阻塞。一旦将一个进程连接到管道的另一端时，情况就会有所改变，进程会从管道中读取数据。使用第二种终端窗口，输入如下命令。

```
[me@linuxbox ~]$ cat < pipe1
```

第一个终端窗口产生的目录列表作为 cat 命令的输出出现在第二个终端窗口。一旦它不处于阻塞状态，第一个终端窗口中的 ls 命令就可以成功完成。

36.5 本章结尾语

到此我们已经结束了所有的学习内容。现在要做的唯一一件事就是练习，练习，再练习。尽管在学习的过程中已经涉及了大量的内容，但是，我们了解的仅仅是命令行方面的皮毛而已。仍然还有成千上万的命令行程序等着去发现和探索。深入地探索/usr/bin 目录内容之后，相信你一定会有所收获！



想必读者已经有过在 Linux 计算机的屏幕上操作图形界面的经历，现在我们开始深入探究 Linux 命令行的强大功能。

本书为读者讲解从在终端上进行第一次击键输入，到使用 Bash 这款最流行的 Linux shell 来编写完整程序的所有内容。在这个过程中，你将学到由数代命令行操作大师流传下来的这一永不过时的技巧：文件导航、环境配置、命令链、正则表达式的模式匹配等。

除了这些实用的知识之外，作者还讲解了这些工具背后的理念，以及桌面 Linux 系统从昔日 UNIX 超级计算机中继承来的丰富功能。

一旦略窥门径，你就会发现命令行是一种与计算机进行通信的最自然、最具表现力的方式。

学完本书之后，你将学会如何：

- 创建和删除文件、文件夹、符号链接；
- 管理系统，包括连网、安装软件包和集成管理；
- 使用标准输入、输出、重定向和管道；
- 使用vi这款世界上最流行的文本编辑器来编辑文件；
- 编写shell脚本，用于自动处理常见的或枯燥的任务；
- 使用cut、paste、grep、patch和sed命令对文本文件进行分割或切块。

William E. Shotts, Jr.作为一名软件开发人员和狂热的Linux用户已经有15年之久。他在软件开发领域有深厚的背景，先后涉及过技术支持、质量保证和文档编写等工作。他还是LinuxCommand.org网站的创始人，该网站是一个Linux教育和宣传网站，以新闻、评论和为人们使用Linux命令行提供广泛支持而见长。

人民邮电出版社-信息技术分社
http://weibo.com/ptpitbooks

ISBN 978-7-115-30745-3
9 787115 307453 >
ISBN 978-7-115-30745-3
定价：69.00 元

美术编辑：王建国

分类建议：计算机/操作系统/Linux
人民邮电出版社网址：www.ptpress.com.cn